
# UNIVERSIDAD NACIONAL DEL SUR

Tesis de Magíster
en Ciencias de la Computación

*Metodologías y Métricas para*
*Desarrollo de Sistemas de Software*

Martín Mauricio Pérez

Bahía Blanca Argentina

2007


# UNIVERSIDAD NACIONAL DEL SUR

Tesis de Magíster
en Ciencias de la Computación

*Metodologías y Métricas para*
*Desarrollo de Sistemas de Software*

Martín Mauricio Pérez

Bahía Blanca Argentina

2007

Fecha de Defensa: ____ / ____ / _____ Nota: _____

# Prefacio

Esta Tesis se presenta como parte de los requisitos para optar al grado académico de Magíster en Ciencias de la Computación, de la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otras. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el Departamento de Ciencias e Ingeniería de la Computación durante el período comprendido entre el mes de abril de 2000 y el mes de diciembre de 2007, bajo la dirección del Dr. Pablo R. Fillottrani, Profesor Adjunto del Departamento de Ciencias e Ingeniería de la Computación.

12 de junio de 2009

Martín M. Pérez

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

# Resumen

El desarrollo de sistemas de software es una disciplina relativamente joven que no ha alcanzado el nivel de madurez de otras ingenierías. Como consecuencia, los proyectos de desarrollo resultan frecuentemente en procesos inestables e impredecibles. Las Ciencias de la Computación han contribuido con conocimiento teórico, pero ciertas áreas necesitan una mayor evolución para poder generar nuevas metodologías de trabajo.

Uno de los aspectos más críticos es, dado el conjunto de metodologías existentes, determinar cuál corresponde utilizar para la construcción de un producto específico. Esta decisión involucra elegir, entre otras cosas, cuáles son los pasos a seguir y qué se espera lograr al finalizar cada una de las distintas etapas. El hecho de definir un proceso de producción de software nos permite repetir los beneficios de un proceso estandarizado. Sin embargo, hay que tener presente que es un proceso que (como toda actividad ingenieril) necesita de técnicas y metodologías precisas y bien definidas, y todavía no se ha logrado caracterizarlas por completo por lo novedoso de la disciplina.

Este trabajo, en primer término, analiza y compara metodologías de desarrollo de sistemas e investiga las distintas áreas de las Ciencias de la Computación sobre las cuales se fundamentan.

La utilización de una metodología para el desarrollo de sistemas es la base para poder realizar la planificación y el control de proyectos. Asociados a estas tareas se encuentran los procesos de medición que se aplican tanto al producto como a los pasos para su construcción. Para cada una de las metodologías aplicadas, el trabajo profundiza sobre las métricas específicas existentes.

# Abstract

The software systems development is a relatively young discipline, and has not reached the same level of maturity of other engineering disciplines. As a consequence, the development projects result in unstable and unpredictable processes. Computer Sciences have contributed with theoretical knowledge, but certain areas need a grater evolution to generate new work methodologies.

One of the more critical aspects is, given the set of existing development methodologies, to determine which of them to use to construct a specific product. This decision involves the choice, among other things, of what are the steps to follow and what is the expected result at the end of each step. The fact of defining a software production process makes possible to repeat the benefits of a standardized process. However, it is necessary to take into account that it is a process that (as every engineering activity) needs precise and well defined techniques and methodologies, and they are not completely characterized because of the discipline novelty.

At a first stage, this work analyzes and compares systems development methodologies and investigates the different Computer Sciences areas on which they are foundated.

The use of a systems development methodology is the base to carry out the projects planning and control. Associated with these tasks, there are measurement processes that apply to product as well as to the steps for their construction. For each of the applied methodologies, this work goes into existing specific metrics in depth.

# Índice general

# Capítulo 1

# Introducción

## 1.1. Contexto

Un *Sistema Informático* utiliza computadoras para almacenar datos, procesarlos y ponerlos a disposición de quien se considere oportuno. Un sistema puede ser tan simple como una persona que utiliza una computadora para organizar datos elementales de una pequeña empresa familiar. Habitualmente, sin embargo, una empresa mediana tiene más de una computadora. Y en el otro extremo, la mayor parte de los sistemas son mucho más complejos, y hacen uso intensivo de recursos informáticos como computadoras, redes, impresoras, y sobre todo el *software*, que en la mayoría de los casos debe ser hecho a la medida de las necesidades de cada empresa.

Boehm define al software como *"el conjunto de programas, procedimientos y documentación asociados a un sistema, y particularmente a un sistema computacional"* [Boe06].

Los sistemas de información tienen muchas cosas en común, la mayoría de ellos están formados por:

- *Personas:* son un componente esencial en cualquier sistema de información, producen y utilizan la información de sus actividades diarias para decidir lo que se debe hacer. Las decisiones pueden ser rutinarias o complejas.

- *Procedimientos:* los sistemas de información deben soportar diversas clases de actividades del usuario, por eso han de establecerse procedimientos que aseguren que los datos correctos llegan a las personas adecuadas en su momento justo.

- *Equipo:* es decir las computadoras y todos los dispositivos necesarios.

En este contexto cobran relevancia cuestiones tales como la *Ingeniería de Software*, que es el nombre que se aplica al conjunto de prácticas actuales para el desarrollo de sistemas, y las *Metodologías de Desarrollo de Software*, que son los diferentes procedimientos, técnicas y ayudas a la documentación para el desarrollo de productos de software.

## 1.2. Ingeniería de Software

Existen muchas definiciones de Ingeniería de Software. Entre ellas pueden citarse:

- Bauer: *"Ingeniería de Software es el establecimiento y uso de firmes principios y métodos de ingeniería para la obtención económica de software fiable y que funcione en máquinas reales"* [Nau68].

- Ghezzi, Jazayeri y Mandrioli: *"La Ingeniería de Software es el campo de las ciencias de la computación que trata con la construcción de sistemas de software, los que son tan grandes y complejos que requieren ser construidos por un equipo o equipos de ingenieros"* [GJM91].

- Boehm: *"Ingeniería de Software es la aplicación de la ciencia y las matemáticas mediante la cual la capacidad de los equipos computacionales se hacen útiles al hombre a través de programas de computadora, procedimientos y la documentación asociada"* [Boe06].

- IEEE: *"Ingeniería del Software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software; es decir, la aplicación de la ingeniería al software"* [Std93].

El origen de la Ingeniería de Software estuvo dado por la denominada "crisis del software" (años 1965 - 1970), caracterizada por el desarrollo inacabable de grandes programas, la ineficiencia, los errores, y un costo impredecible, como consecuencia de una falta de formalismo y metodología, de herramientas de soporte, y de una administración eficaz.

Actualmente está surgiendo una gran expectativa ante la evolución de la Ingeniería del Software, al ir apareciendo nuevos métodos y herramientas formales que

prometen un planteamiento de ingeniería en el proceso de elaboración de software. Dicho planteamiento vendrá a paliar la demanda creciente por parte de los usuarios, permitiendo dar respuesta a los problemas de administración, calidad, productividad y fácil mantenimiento.

Hoy en día el software tiene un doble papel. Es un producto, pero simultáneamente es el vehículo para hacer entrega de un producto [MBA06]. Como *producto* permite el uso del hardware, como por ejemplo una computadora personal o un teléfono móvil. Como *vehículo* utilizado para hacer entrega del producto, actúa como base de control, por ejemplo un sistema operativo, o un sistema gestor de redes. El software hace entrega de lo que se considera como el producto más importante del siglo veintiuno, la *información*. El software transforma datos personales para que sean más útiles en un entorno local, gestiona información comercial para mejorar la competitividad, proporciona el acceso a redes a nivel mundial, y ofrece el medio de adquirir información en todas sus formas.

Actualmente se considera la Ingeniería del Software como una nueva área de la ingeniería, y la profesión de ingeniero informático es una de las más demandadas. La ingeniería del software trata áreas muy diversas de la informática y de las ciencias de la computación, aplicables a un amplio espectro de campos, tales como negocios, investigación científica, medicina, producción, logística, banca, meteorología, derecho, redes, entre otras muchas.

El *Ingeniero de Software* es una persona que trabaja en equipo, que sabe que lo que realiza es un componente que se debe combinar con otros para formar un sistema. Es consciente de que el componente de software que diseña debe poseer los principios de la Ingeniería del Software para que el sistema final sea satisfactorio.

Sin embargo, es frecuente que en la práctica diaria profesional no se incluya prácticamente ninguna de las recomendaciones más elementales de la Ingeniería de Software. De hecho, las evaluaciones de los procesos productivos de software realizadas a raíz de los modelos de procesos de software confirman que el desarrollo de software suele estar básicamente en estado caótico. Y no sólo en pequeñas empresas de países como Argentina, sino en grandes proyectos en naciones como EE UU y Japón.

La formalización del proceso de desarrollo se define como un marco de referencia denominado *ciclo de desarrollo del software* o *ciclo de vida del desarrollo del software*. Se puede describir como, "el período de tiempo que comienza con la decisión de

desarrollar un producto de software y finaliza cuando éste se ha entregado" [MBA06]. Este ciclo, por lo general incluye las fases:

- Requisitos
- Diseño
- Implementación
- Prueba
- Instalación
- Aceptación

Un *modelo de proceso de software* o *modelo de ciclo de vida* es una representación simplificada de un proceso de software que conlleva una estrategia global para abordar el desarrollo de software. Existen diferentes modelos de proceso, entre ellos:

- Codificar y corregir (*code-and-fix*)
- Desarrollo en cascada
- Desarrollo evolutivo
- Desarrollo formal de sistemas
- Desarrollo basado en reutilización
- Desarrollo incremental
- Desarrollo en espiral

## 1.3. Metodologías para Desarrollo de Software

La metodología de desarrollo de software define *quién* está haciendo *qué*, *dónde* y *cómo* para lograr un cierto objetivo [JBR99]. En Ingeniería de Software, el objetivo es construir un producto de software o mejorar uno existente. Un proceso efectivo provee lineamientos para el desarrollo eficiente de software de calidad. Captura y presenta las mejores prácticas que el estado actual del arte permite. En consecuencia, reduce

el riesgo y aumenta la predecibilidad. El efecto global es promover una visión y una cultura comunes.

Un *proceso* es necesario como guía para todos los participantes - clientes, usuarios, desarrolladores y directores ejecutivos. El proceso debe ser lo mejor que la industria sea capaz de reunir en el momento actual, y además debe estar ampliamente disponible para que todos los interesados puedan comprender su rol en el desarrollo bajo consideración.

Un proceso de desarrollo de software, además, debería ser capaz de evolucionar a lo largo de muchos años. Durante esta evolución debería limitar su alcance, en cada momento, a las realidades que las tecnologías, herramientas, personas, y estándares de la organización permitan.

Las *Metodologías de Desarrollo de Software* son un conjunto de procedimientos, técnicas y ayudas a la documentación para el desarrollo de productos de software [MBA06]. Una metodología indica paso a paso todas las actividades a realizar para lograr el producto informático deseado, indicando además qué personas deben participar en el desarrollo de las actividades y qué papel deben tener. Además detallan la información que se debe producir como resultado de una actividad y la información necesaria para comenzarla.

Actualmente es imprescindible considerar los *riesgos*, aunque habitualmente las empresas no han sido conscientes de los riesgos inherentes al procesamiento de la información mediante computadoras, a lo que han contribuido, a veces, los propios responsables de informática, que no han sabido explicar con la suficiente claridad las consecuencias de una política de seguridad insuficiente o incluso inexistente. Por otro lado, debido a una cierta deformación profesional en la aplicación de los criterios de costo/beneficio, el directivo desconocedor de la informática no acostumbra a autorizar inversiones que no lleven implícito un beneficio demostrable y tangible.

Las *técnicas* indican cómo se debe realizar una actividad técnica determinada identificada en la metodología. Combina el empleo de modelos o representaciones gráficas junto con procedimientos detallados. Se debe tener en cuenta que una técnica determinada puede ser utilizada en una o más actividades de la metodología de desarrollo de software. Además se debe tener mucho cuidado cuando se quiere cambiar una técnica por otra.

Lo más importante en una empresa de desarrollo de software es disponer de *per-*

*sonas* calificadas, aunque ello no asegura el éxito en la consecución de los objetivos propuestos. Siempre existe el peligro de falta de conjunción, producida por la manera personal de desarrollar el software de cada individuo, y la imposibilidad de un auténtico trabajo en equipo. Los mejores informáticos necesitan un entorno disciplinado y estructurado para poder realizar un trabajo en equipo, para lograr productos de alta calidad.

Los programadores tradicionales argumentan que la aplicación de una metodología supone una gran carga. Esto puede ser cierto, pero si no se emplea una metodología pueden surgir los siguientes problemas:

- Resultados impredecibles
- Detección tardía de errores
- La introducción de nuevas herramientas afectará perjudicialmente al proceso
- Cambios en la organización también afectarán al proceso
- Resultados distintos con nuevas clases de productos

Existen innumerables metodologías de desarrollo, algunas de las cuales se listan a continuación, por sólo mencionar unas pocas:

- Desarrollo Convencional (sin metodología)
- Desarrollo Estructurado (metodologías orientadas a la función, a los datos, o mixtas)
- Desarrollo Orientado a Objetos (muchas metodologías diferentes)
- Metodologías Ágiles (Extreme Programming, Scrum, Evo, Crystal Methods, etc...)

En el momento de adoptar una metodología, deben considerarse una serie de requisitos deseables que deben cumplir:

- La metodología debe ajustarse a los objetivos.
- La metodología debe cubrir el ciclo entero de desarrollo de software.
- La metodología debe integrar las distintas fases del ciclo de desarrollo.

- La metodología debe incluir la realización de validaciones.
- La metodología debe soportar la determinación de la exactitud del sistema a través del ciclo de desarrollo.
- La metodología debe ser la base de una comunicación efectiva.
- La metodología debe funcionar en un entorno dinámico orientado al usuario.
- La metodología debe especificar claramente los responsables de los resultados.
- La metodología debe poder emplearse en un entorno amplio de proyectos de software.
- La metodología se debe de poder enseñar.
- La metodología debe estar soportada por herramientas CASE.
- La metodología debe soportar la eventual evolución del sistema.
- La metodología debe contener actividades conducentes a mejorar el proceso de desarrollo de software.

## 1.4. Objetivos

En base a lo anteriormente expuesto, el presente trabajo tiene la intención de analizar y comparar tres metodologías diferentes de desarrollo de sistemas, e investigar las distintas áreas de las Ciencias de la Computación sobre las cuales se fundamentan. Las metodologías que se investigarán son:

- *Ciclo de Vida Espiral*
- *Metodologías para Desarrollos Basados en Objetos*
- *Métodos Formales*

La utilización de una metodología para el desarrollo de sistemas es la base para poder realizar la planificación y control de proyectos. Asociado a estas tareas se encuentran los procesos de medición que se aplican tanto al producto como a los pasos

para su construcción. En este marco, este trabajo también se propone estudiar sobre las distintas *métricas* existentes para analizar los atributos internos y externos del producto, la utilización de los resultados de los procesos de medición como predictores de costo y esfuerzo, y las mediciones que se realizan para el ciclo de vida. Para cada una de las metodologías mencionadas, se profundiza sobre las métricas específicas.

Finalmente, se elaboran conclusiones sobre cada una de las metodologías estudiadas, destacando sus fortalezas y debilidades, y comentando qué metodología es más adecuada en función del tipo de sistema a desarrollar. Se realiza además un estudio comparativo entre las distintas metodologías, y entre las métricas específicas de cada una.

## 1.5. Organización

El presente trabajo se organiza de la siguiente manera:

- En el Capítulo 2 se describe en detalle el Ciclo de Vida en Espiral, y las cuestiones relacionadas con la gestión y supervisión de riesgos aplicada a esta metodología.
- En el Capítulo 3 se presentan las particularidades de las Metodologías Orientadas a Objetos en general, investigando luego el Método Unificado de Desarrollo en particular, y UML como lenguaje de especificación.
- En el Capítulo 4 se estudian en detalle las principales características de los Métodos Formales de Desarrollo, para luego concentrarse en los lenguajes y métodos que soportan este tipo de desarrollo, profundizando en el Método RAISE y RSL como lenguaje de especificación.
- Para cada una de las tres metodologías presentadas, al final de cada capítulo, se estudian las métricas específicas que pueden extraerse de sus procesos o productos.
- Finalmente, en el Capítulo 5 se elaboran las conclusiones surgidas del trabajo.

# Capítulo 2

# Ciclo de Vida en Espiral

El *desarrollo en espiral*, originalmente propuesto por Boehm [Boe88], es una familia de procesos de desarrollo de software que se caracteriza por iterar repetidamente un conjunto de procesos de desarrollo elementales y manejar los riesgos de forma que se reducen activamente. Acompaña la naturaleza iterativa de la construcción de prototipos con aspectos controlados y sistemáticos del modelo lineal secuencial. Proporciona el potencial para el desarrollo rápido de versiones incrementales del software. En el modelo en espiral, el software se desarrolla en una serie de versiones incrementales: durante las iteraciones iniciales, la versión incremental puede ser un modelo en papel o un prototipo, durante las iteraciones finales, se producen versiones cada vez más completas del sistema.

El modelo en espiral se presentó como una alternativa para mejorar la situación de los modelos de proceso de software. Su principal característica distintiva es que crea un enfoque *conducido por el riesgo* en lugar de un proceso principalmente conducido por los documentos o por el código. Incorpora muchas de las fortalezas de otros modelos, a la vez que resuelve muchas de sus dificultades.

Al contrario de otros ciclos de vida clásicos, que finalizan cuando el proyecto se concluye, el ciclo de vida en espiral puede ser adoptado para toda la vida del proyecto. El comienzo de cada nuevo ciclo de la espiral puede ser utilizado para crear nuevas versiones del programa o desarrollar nuevos productos que tienen como base el que ya existía. En cualquier caso el comienzo del desarrollo está aquí, por lo tanto la espiral permanece en funcionamiento hasta que el software queda obsoleto.

## 2.1. El Modelo de Desarrollo en Espiral

La Figura 2.1 es el diagrama original del modelo en espiral publicado por Boehm [Boe88]. Captura las características principales del modelo en espiral: ingeniería concurrente cíclica; determinación de proceso y producto conducida por los riesgos; crecimiento de un sistema mediante experimentación y elaboración conducidas por los riesgos; y disminución de los costos de desarrollo mediante la eliminación temprana de alternativas inviables y la evasión del trabajo redundante.

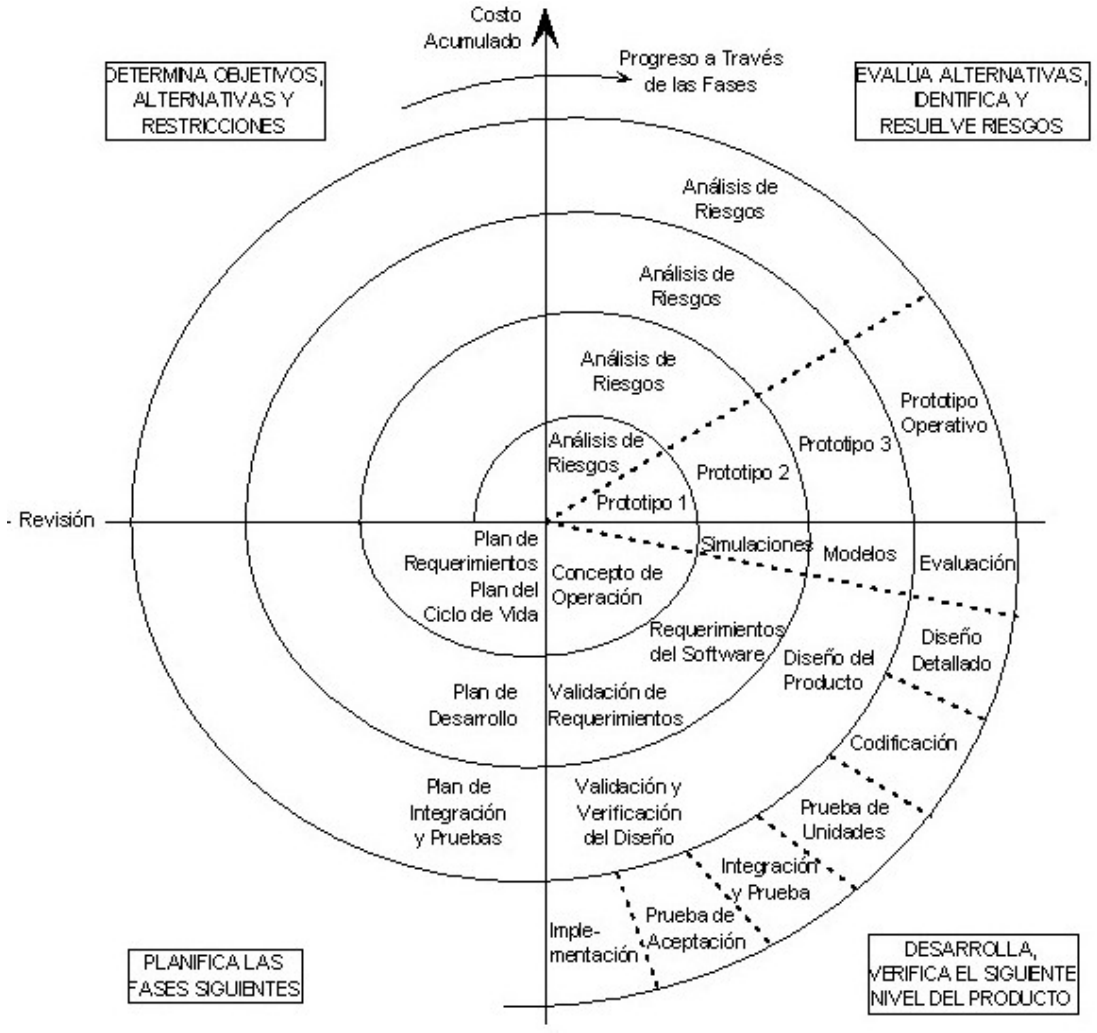


Figura 2.1: Diagrama Original del Desarrollo en Espiral

Como resultado del planeamiento y el análisis del riesgo, diferentes proyectos pueden elegir diferentes procesos. Es decir, el modelo en espiral es en realidad un gene-

rador de modelos de proceso conducido por los riesgos, en el que diferentes patrones de riesgos pueden llevar a elegir procesos incrementales, en cascada, prototipado evolutivo, u otros subconjuntos de los elementos de procesos en el diagrama del modelo en espiral.

Sin embargo, por varias razones, el modelo en espiral no es comprendido universalmente. Por ejemplo, la Figura 2.1 contiene algunas simplificaciones que han causado la propagación de algunas interpretaciones erróneas acerca del modelo en espiral. Las equivocaciones más significativas, y que hay que evitar, son: que la espiral es sólo una secuencia de incrementos en cascada; que todo en el proyecto sigue una única secuencia en espiral; que todos los elementos del diagrama deben visitarse en el orden indicado; y que no puede haber retrocesos para revisar decisiones previas. Además de estos malentendidos, otros procesos similares –pero peligrosamente distintos– han sido tomados como procesos en espiral.

Para facilitar la comprensión y el uso efectivo del modelo en espiral, se caracterizará más precisamente al mismo, comenzando con una definición simple que captura la esencia del modelo [Boe00a]:

> El modelo de desarrollo en espiral es un **generador de modelos de proceso** conducido por los **riesgos**. Se usa para guiar la ingeniería concurrente de sistemas de software. Tiene dos características principales que lo distinguen: una es un enfoque **cíclico** para el crecimiento incremental del grado de definición e implementación de un sistema mientras se disminuye su grado de riesgo; la otra es un conjunto de **hitos** que le aseguran a los interesados el logro de soluciones factibles y mutuamente satisfactorias.

Los **riesgos** son situaciones o posibles eventos que pueden causar que un proyecto no logre sus objetivos. Su impacto puede ir desde lo trivial a lo fatal, y su probabilidad puede ir desde lo cierto a lo improbable. Un plan de gestión de riesgos los enumera y les da prioridades por su grado de importancia, que se mide como una combinación de impacto y probabilidad de cada uno. Por cada riesgo, el plan también establece una estrategia de mitigación para ocuparse del mismo. Por ejemplo, el riesgo de que la tecnología no esté lista puede mitigarse mediante una implementación prototípica apropiada en los primeros ciclos de la espiral.

Un **modelo de proceso** responde a dos preguntas principales:

- ¿Qué debería hacerse ahora?
- ¿Por cuánto tiempo debería continuar?

Bajo el modelo en espiral, las respuestas a estas preguntas son conducidas por las consideraciones de riesgos y varían de un proyecto a otro, y a veces de un ciclo de la espiral al siguiente. Cada elección de respuestas genera un modelo de proceso diferente. Al comienzo de un ciclo, todos los interesados en el éxito del proyecto deben participar concurrentemente en la revisión de riesgos y en la consecuente elección del modelo de proceso para el proyecto.

El modelo en espiral abarca a otros modelos. Si no hay un gran riesgo en la recolección de datos, se convierte en un sencillo modelo en cascada, mientras que en sistemas de más compleja definición de requisitos, como en los sistemas altamente interactivos, el sistema puede considerarse como un sistema evolutivo, con preponderancia en las actividades de prototipado, y con un tercer cuadrante relativamente pequeño.

La naturaleza **cíclica** del modelo en espiral se ilustró en la Figura 2.1. Se puede ver el proceso de desarrollo como un diagrama polar, donde la dimensión angular corresponde al progreso de un proyecto a través del tiempo, y la dimensión radial corresponde al costo acumulado del proyecto. Cuando el progreso ha avanzado 360 grados, se ha completado un **ciclo**.

Cada ciclo está sujeto a cuatro **fases**:

1. *Identificación:* determinar objetivos, alternativas, y restricciones del ciclo.
2. *Evaluación:* evaluar alternativas, analizar riesgos frente a los objetivos, y posiblemente resolver los riesgos mediante construcción de prototipos, etc.
3. *Ingeniería:* desarrollo y verificación del producto intermedio. Aquí se puede usar un modelo en cascada.
4. *Planificación* de la próxima iteración.

Entre las fases de Evaluación e Ingeniería se construyen, ejecutan, y evalúan diversos prototipos del producto final. Estos prototipos pueden ser de varias clases:

- Ilustrativo: se enfatiza la interfaz con el usuario.
- Funcional: se enfatizan las funciones más importantes, dadas por los riesgos identificados.

- Exploratorio: se enfatiza la interfaz con otros sistemas.
- Simulación: se modela un sistema del mundo real.

Los **hitos** conducen el progreso de la espiral hacia su conclusión y ofrecen un medio para comparar el progreso entre un proyecto en espiral y otro.

### 2.1.1. Un ciclo típico de la espiral

Cada ciclo de la espiral [Boe88] comienza con la *identificación* de

- los objetivos de la porción del producto que se está elaborando (desempeño, funcionalidad, capacidad de adaptarse al cambio, etc.);
- los medios alternativos para implementar esa porción del producto (diseño A, diseño B, reusar, comprar, etc.); y
- las restricciones impuestas sobre la aplicación de las alternativas (costo, tiempo, interfaz, etc.).

El próximo paso es *evaluar* las alternativas en relación con los objetivos y restricciones. Con frecuencia, este proceso identifica áreas de incertidumbre que son fuentes significativas de riesgos del proyecto. En ese caso, la próxima etapa debería incluir la formulación de una estrategia efectiva para resolver las fuentes de riesgo. Esto puede involucrar prototipado, simulación, pruebas de referencias, cuestionarios al usuario, modelado analítico, o combinaciones de éstas y otras técnicas de resolución de riesgos.

Una vez evaluados los riesgos, la próxima etapa está determinada por los riesgos remanentes relativos. Si los riesgos de desempeño o de interfaz de usuario dominan fuertemente el desarrollo del programa o los riesgos internos de control de la interfaz, el próximo paso puede ser un desarrollo evolutivo: un esfuerzo mínimo para especificar la naturaleza general del producto, un plan para el siguiente nivel de prototipado, y el desarrollo de un prototipo más detallado para continuar resolviendo los riesgos más importantes.

Si el prototipo es operativamente útil y lo bastante robusto como para servir de base de bajo riesgo para la futura evolución del producto, los pasos siguientes conducidos por el riesgo deberían ser una serie de prototipos evolutivos, yendo hacia la derecha en la Figura 2.1. En este caso no se ejercita la escritura de especificaciones. La

consideración de riesgos lleva así a un proyecto que implementa sólo un subconjunto de todos los potenciales pasos del modelo.

Por el contrario, si los prototipos previos ya han resuelto todos los riesgos de desempeño o de interfaz de usuario, y predominan los riesgos de desarrollo del programa o los de control de la interfaz, el próximo paso sigue el enfoque básico en cascada (concepto de operación, requerimientos del software, diseño preliminar, etc.) modificado para incorporar el desarrollo incremental. Cada nivel de especificación de software en la figura es seguido luego por una etapa de validación y la preparación de planes para el siguiente ciclo. En este caso no se ejercitan las opciones de prototipado, simulación, modelado, etc., llevando así al uso de un subconjunto diferente de etapas.

La elección conducida por los riesgos de diferentes subconjuntos de pasos del modelo en espiral permite que el modelo se adecue a cualquier mezcla apropiada de enfoques de desarrollo de software orientados a la especificación, al prototipo, a la simulación, a la transformación automática, etc. En tales casos, se elige la estrategia apropiada considerando la magnitud relativa de los riesgos del programa y la efectividad relativa de las distintas técnicas para resolver los riesgos. De manera similar, las consideraciones de gestión de riesgos pueden determinar la cantidad de tiempo y esfuerzo que se debería dedicar a otras actividades del proyecto tales como planificación, gestión de configuración, control de calidad, verificación formal y pruebas. En particular, las especificaciones conducidas por los riesgos pueden tener varios grados de completitud, formalidad y granularidad, dependiendo de los riesgos relativos de especificar de más o de menos.

Una característica importante del modelo en espiral es que cada ciclo se completa con una *revisión* que involucra a las personas u organizaciones principalmente interesadas en el producto. Esta revisión cubre todos los productos desarrollados durante el ciclo previo, incluyendo los planes para el próximo ciclo y los recursos requeridos para llevarlos a cabo. El principal objetivo de esta revisión es asegurar que todas las partes interesadas estén mutuamente comprometidas con la aproximación de la próxima fase.

Los *planes* para etapas subsiguientes también pueden incluir una partición del producto en incrementos, para que sucesivos desarrollos o productos sean efectuados por organizaciones o personas individuales, agregando así una tercera dimensión al concepto presentado en la Figura 2.1.

### 2.1.2. Inicio y terminación de la espiral

Al considerar esta presentación del modelo en espiral surgen cuatro preguntas fundamentales:

1. ¿Cómo comienza la espiral?
2. ¿Cómo se sale de la espiral cuando lo apropiado es terminar tempranamente un proyecto?
3. ¿Por qué la espiral termina en forma tan abrupta?
4. ¿Qué sucede con la mejora (o mantenimiento) del software?

La respuesta a estas preguntas implica observar que el modelo en espiral se puede aplicar igualmente bien a esfuerzos de desarrollo y de mejora. En cualquier caso, la espiral comienza con la hipótesis de que una misión operativa en particular (o un conjunto de misiones) podría mejorarse mediante un esfuerzo de software. El proceso en espiral supone entonces una prueba de esa hipótesis: en cualquier momento, si la hipótesis no supera la prueba (por ejemplo, si las demoras causan que un producto de software pierda su ventana de mercado, o si aparece un producto comercial superior), la espiral se termina. Si no, termina con la instalación del software nuevo o modificado, y la hipótesis se contrasta mediante la observación del efecto sobre la misión operativa.

Comúnmente, la experiencia con la misión operativa lleva a otras hipótesis sobre mejoras en el software, y así se inicia una nueva espiral de mantenimiento para probar esas hipótesis. El comienzo, la finalización, y la iteración de las tareas y productos de los ciclos previos están así definidos implícitamente en el modelo en espiral.

### 2.1.3. Concepto de riesgo en un proyecto informático

En general, cualquier decisión humana está basada en la probabilidad de que se produzca un escenario favorable. En un proyecto informático, se denomina riesgo a un factor no controlado totalmente que afecta al desarrollo o a su gestión, poniendo en peligro el éxito del proyecto frente a sus usuarios/clientes. Por ejemplo: no disponibilidad de personal adecuado, planificación y presupuestos poco realistas, desarrollo de funciones de software equivocadas, desarrollo de una incorrecta interfaz de

usuario, cambios continuos en los requisitos, imposibilidad técnica de satisfacer una determinada funcionalidad, etc.

Un típico riesgo es la inadecuación de la interfaz de usuario al tipo de uso que se quiere dar al sistema. En un sistema que pretende ser de uso público, por ejemplo, en el que se requerirá una gran facilidad de manejo para todos, un fallo en la interfaz sería un desastre para el sistema completo. Esto se resuelve en la etapa que corresponda mediante la construcción de bocetos y prototipos parciales que habrán de probarse ampliamente antes de abordar su desarrollo.

En un proyecto, si bien la probabilidad de los escenarios desfavorables es baja, sus efectos en caso de producirse podrían ser graves. Algunos riesgos típicos son:

- ¿Se estarán entendiendo bien las necesidades del cliente?
- ¿Será realmente idóneo el programador al cual se le encargó la construcción de tal o cual módulo?
- ¿Será este paquete de aplicación tan adecuado como se cree?

Respecto a la primera pregunta, si se han efectuado suficientes entrevistas y estudiado bien la situación actual y el usuario ha dado su aprobación a los planes, la probabilidad de un escenario positivo es alta, pero no de 100 %. Si se siguiera confiadamente adelante y sólo al terminar el proyecto se hiciera evidente que los requerimientos no eran los adecuados, el costo sería enorme: habría que rehacer el sistema.

En todo proyecto es necesario corroborar –a través de un *Análisis de Riesgos*– que la probabilidad de que se den los escenarios supuestos es suficientemente alta. Independientemente de la etapa y del grado de avance, las actividades de planificación, análisis de riesgos, ejecución y control son recurrentes. No se deben confundir con etapas del proyecto. Son actividades que se ejecutan con creciente grado de intensidad y profundidad a medida que se avanza.

El análisis de riesgos trata de reducir la incertidumbre o de confirmar que los escenarios positivos conservan la alta probabilidad supuesta *a priori*. A diferencia del tradicional modelo en cascada, que supone un avance quemando etapas, el modelo en espiral presenta al desarrollo de sistemas como un continuo que cruza los tópicos de avance con un conjunto de actividades tipo, lo cual es más coherente con la realidad.

Una consecuencia de la flexibilidad del modelo en espiral es que el desarrollador debe tomar decisiones en varias etapas del proceso de desarrollo. Esta toma de

decisiones supone riesgos. El modelo en espiral guía al desarrollador a posponer la elaboración detallada de los elementos de software de menor riesgo y a evitar ir demasiado en profundidad en el diseño hasta que los elementos de mayor riesgo del diseño estén estabilizados. La gestión de riesgos requiere en las primeras fases del desarrollo de un sistema de software, técnicas de resolución de riesgos tales como prototipado y simulación. El modelo en espiral permite la incorporación de técnicas de prototipado como una opción de reducción de riesgos en cualquier fase de desarrollo y actividades de control de riesgos.

La gestión de riesgos implica los siguientes pasos:

- Técnicas de valoración de riesgos
  - La *identificación* de riesgos produce una lista de elementos de riesgo específicos del proyecto que probablemente comprometan el éxito del proyecto.
  - El *análisis* de riesgos cuantifica la probabilidad de pérdida y la magnitud de pérdida para cada elemento de riesgo identificado.
  - La *prioridad* de los riesgos produce una lista gradual de los elementos de riesgo de acuerdo a su severidad.
- Técnicas de control de riesgos
  - La *planificación de la gestión* de riesgos ayuda a localizar cada elemento de riesgo. También incluye la coordinación entre los planes individuales de cada elemento de riesgo con el plan del proyecto global.
  - La *resolución* de riesgos produce una situación en la que se eliminan o se resuelven los elementos de riesgo.
  - El *seguimiento* de riesgos implica el rastreo del progreso del proyecto hacia la resolución de sus elementos de riesgo y hacia la toma de acciones correctivas cuando sean necesarias.

En el modelo en espiral, las actividades del segundo cuadrante requieren que los objetivos definidos en el primer cuadrante sean evaluados con respecto a las alternativas y a las limitaciones identificadas. El resultado de este análisis es una identificación, cuantificación y obtención de un lista gradual de los riesgos asociados con cada enfoque alternativo.

## 2.2. Evaluación del Modelo en Espiral

### 2.2.1. Ventajas

La principal ventaja del modelo en espiral es que su rango de opciones adopta las características buenas de los modelos existentes de proceso de software, mientras que su enfoque conducido por riesgos evita muchas de sus dificultades [Boe88]. En situaciones apropiadas, el modelo en espiral equivale a uno de los modelos de proceso existentes. En otras situaciones, provee una guía para elegir la mejor combinación de enfoques existentes para un proyecto dado.

Las condiciones principales bajo las cuales el modelo en espiral se vuelve equivalente a otros modelos de proceso se resumen a continuación:

- Si un proyecto tiene bajo riesgo en áreas tales como lograr una interfaz de usuario equivocada o no alcanzar estrictos requerimientos de desempeño, y si tiene alto riesgo en la predicción y control de presupuesto y planificación, entonces estas consideraciones del riesgo conducen a que el modelo en espiral sea equivalente al modelo en cascada.

- Si los requerimientos de un producto de software son muy estables, y si la presencia de errores en el producto de software constituye un alto riesgo para la misión que cumple, entonces estas consideraciones del riesgo llevan a que el modelo en espiral se parezca al modelo de dos patas de especificación precisa y desarrollo deductivo formal del programa.

- Si un proyecto tiene bajo riesgo en áreas tales como pérdida de predecibilidad y control de presupuesto y planificación, tropiezos con problemas de integración de sistemas grandes, o copias con pérdidas de información, y si tiene un riesgo alto en áreas como equivocar la interfaz de usuario o los requerimientos de soporte a las decisiones del usuario, entonces estas consideraciones del riesgo conducen a que el modelo en espiral sea equivalente al modelo de desarrollo evolutivo.

- Si se dispone de capacidades de generación automática de software, entonces el modelo en espiral las integra como opciones para prototipado rápido o para

aplicación del modelo de transformación, dependiendo de las consideraciones del riesgo involucradas.

- Si los elementos de alto riesgo de un proyecto incluyen una combinación de los ítems de riesgo listados anteriormente, entonces el enfoque en espiral reflejará una combinación apropiada de los modelos de proceso anteriores. Al hacerlo, sus características de evasión de riesgos generalmente evitarán las dificultades de los otros modelos.

El modelo en espiral tiene varias ventajas adicionales, que se resumen a continuación [Boe88]:

*Pone atención tempranamente en las opciones que involucran el reuso de software existente.* Las etapas que realizan identificación y evaluación de alternativas fomentan estas opciones.

*Incluye una preparación para la evolución y el crecimiento del ciclo de vida, y para los cambios en el producto de software.* Las principales fuentes de cambios en el producto se incluyen en los objetivos del producto, y los enfoques de ocultamiento de información son alternativas atractivas de diseño arquitectónico, ya que reducen el riesgo de no poder incluir los objetivos de carga del producto.

*Provee un mecanismo para incorporar objetivos de calidad del software dentro del desarrollo del producto de software.* Este mecanismo deriva del énfasis en identificar todos los tipos de objetivos y restricciones durante cada ciclo de la espiral.

*Se concentra en la eliminación temprana de errores y alternativas poco atractivas.* Las etapas de análisis de riesgos, validación y compromiso cubren estas consideraciones.

*Por cada una de las fuentes de gastos en recursos y actividad del proyecto, responde a la pregunta clave, "¿cuánto es suficiente?".* Puesto de otro modo, "¿cuánto análisis de requerimientos, planificación, gestión de configuración, control de calidad, pruebas, verificación formal, etc., debería hacer un proyecto?". Usando el enfoque conducido por riesgos, se puede ver que la respuesta no es la misma para todos los proyectos y que el nivel apropiado de esfuerzo se determina por el nivel de riesgo en que se incurre por no hacer suficiente.

*No utiliza enfoques separados para el desarrollo de software y para la mejora (o mantenimiento) de software.* Este aspecto ayuda a evitar el estatus de "ciudadanos de

segunda clase" que con frecuencia se asocia al mantenimiento de software. También ayuda a evitar muchos de los problemas que comúnmente suceden cuando las tareas de mejora de alto riesgo se abordan en la misma forma que las tareas de mantenimiento de rutina.

*Provee un encuadre viable para el desarrollo integrado de hardware-software.* La atención a la gestión de riesgos y a la eliminación temprana y barata de alternativas poco atractivas es igualmente aplicable tanto al hardware como al software.

### 2.2.2. Dificultades

El modelo en espiral completo puede aplicarse con éxito en muchas situaciones, pero antes de que pueda verse como un modelo maduro y universalmente aplicable, se deberían solucionar algunas dificultades [Boe88]. Los tres desafíos principales son: corresponderse con el software por contrato, confiar en expertos en la evaluación de riesgos, y una mayor necesidad de elaboración en los pasos del modelo en espiral.

***Corresponderse con el software por contrato.*** El modelo en espiral en general funciona bien en desarrollos internos de software, pero necesita más trabajo para que se corresponda con el mundo de la adquisición de software por contrato.

Los desarrollos internos de software tienen mucha flexibilidad y libertad para incluir los compromisos etapa por etapa, para diferir compromisos con opciones específicas, para establecer mini espirales que resuelvan elementos del camino crítico, para ajustar niveles de esfuerzo, o para incluir prácticas tales como prototipado, desarrollo evolutivo, o diseño al costo. En la adquisición de software por contrato lleva más tiempo lograr ese grado de flexibilidad y libertad sin perder responsabilidad y control, y más tiempo definir contratos cuyos entregables no están bien especificados de antemano.

Recientemente se progresó mucho en establecer mecanismos de contrato más flexibles, como el uso de contratos competitivos para la definición de conceptos, el uso de contratos de nivel de esfuerzo y de premios para el desarrollo evolutivo, y el uso de contratos de diseño al costo. Aunque estos en general han sido exitosos, los procedimientos para usarlos todavía deben trabajarse más, hasta el punto en que los encargados de la adquisición se sientan completamente cómodos con su uso.

***Confiar en expertos en la evaluación de riesgos.*** El modelo en espiral confía mucho en la capacidad de los desarrolladores de software para identificar y

manejar las fuentes de riesgos del proyecto.

Un buen ejemplo de esto es la especificación conducida por riesgos del modelo en espiral, que describe los elementos de alto riesgo con un elevado nivel de detalle y deja los elementos de bajo riesgo para que sean elaborados en etapas posteriores; en ese momento existe menor riesgo de ruptura.

Sin embargo, un equipo de desarrolladores con poca experiencia también puede producir una especificación con un patrón diferente de variación en su nivel de detalle: gran elaboración de detalles para los elementos de bajo riesgo bien comprendidos, y poca elaboración de los elementos de alto riesgo, menos comprendidos. A menos que haya una revisión intuitiva de esa especificación por parte de personal experto en desarrollo o adquisición, este tipo de proyecto dará una ilusión de progreso durante un período en el cual en realidad se dirige al desastre.

Otra inquietud es que una especificación conducida por riesgos también será dependiente de las personas. Por ejemplo, un diseño producido por un experto puede ser implementado por personas no expertas. En este caso el experto, que no necesita gran cantidad de documentación detallada, debe producir la suficiente documentación adicional como para que los no expertos no se desvíen del camino. Quienes revisan la especificación deben también ser sensibles a estos asuntos.

Con un enfoque convencional, conducido por documentos, el requerimiento de llevar todos los aspectos de la especificación a un nivel uniforme de detalle elimina algunos problemas potenciales y permite una revisión adecuada de algunos aspectos por parte de inexpertos. Pero también produce una gran pérdida de tiempo en los escasos expertos, que deben descubrir los asuntos críticos entre una gran masa de detalles no críticos.

***Necesidad de una mayor elaboración en los pasos del modelo en espiral.*** En general, los pasos del proceso del modelo en espiral necesitan elaboración adicional para asegurar que todos los participantes del desarrollo del software están operando en un contexto consistente.

Algunos ejemplos son la necesidad de definiciones más detalladas de la naturaleza de las especificaciones e hitos del modelo en espiral, la naturaleza y objetivos de las revisiones del modelo en espiral, técnicas para estimar y sincronizar planes, y la naturaleza de los indicadores de estado y procedimientos de seguimiento de costo frente a progreso del modelo en espiral. Existe también una necesidad de lineamientos

y listas de comprobación para identificar las fuentes más probables de riesgos del proyecto y las técnicas de resolución de riesgos más efectivas para cada fuente de riesgo.

Las personas más experimentadas pueden usar con éxito un enfoque en espiral sin estas elaboraciones. Sin embargo, para usos a gran escala en situaciones donde la gente tiene niveles de experiencia diferentes, los niveles agregados de elaboración son importantes para asegurar una interpretación y un uso consistente del enfoque en espiral a lo largo del proyecto.

Los esfuerzos para refinar y aplicar el modelo en espiral se han concentrado en crear una disciplina de gestión de riesgos de software, incluyendo técnicas de identificación, análisis, priorización y planificación de la gestión de los riesgos, y de seguimiento de los elementos de riesgo. Un ejemplo de esto es el plan de gestión de riesgos, que se introduce a continuación.

### 2.2.3. Implicaciones: el Plan de Gestión de Riesgos

Incluso si una organización no está lista para adoptar la estrategia espiral en su totalidad, una técnica característica que se puede adaptar fácilmente a cualquier modelo de ciclo de vida provee muchos de los beneficios del enfoque en espiral. Se trata del *Plan de Gestión de Riesgos* [Boe88], que se resume en los siguientes pasos:

1. Identificar los 10 elementos de riesgo más importantes del proyecto.
2. Presentar un plan para resolver cada elemento de riesgo.
3. Actualizar mensualmente la lista de los elementos de riesgo, plan y resultados.
4. Resaltar el estado de los elementos de riesgo en revisiones mensuales del proyecto (y compararlos con los estados y ubicación en la lista de prioridades del mes anterior).
5. Iniciar las acciones correctivas adecuadas.

Este plan asegura básicamente que cada proyecto hace una identificación temprana de sus elementos de riesgo más importantes, desarrolla una estrategia para resolverlos, identifica y establece una agenda para resolver nuevos elementos de riesgo a medida que aparecen, y resalta el progreso frente a lo planificado en las revisiones mensuales.

En conjunto, el Plan de Gestión de Riesgos y el creciente conjunto de técnicas de gestión de riesgos de software proveen las bases para adaptar los conceptos del modelo en espiral a los procedimientos más establecidos de desarrollo y adquisición de software.

## 2.3. Los Invariantes y sus Variantes

Los ciclos del modelo en espiral siempre presentan seis características [Boe00a]:

1. Determinación de artefactos concurrente, más que secuencial.
2. Consideración en cada uno de los ciclos de la espiral de los principales elementos de la misma.
   - Objetivos y restricciones de los interesados más importantes.
   - Alternativas de producto y de proceso.
   - Identificación y resolución de riesgos.
   - Revisión por parte de los interesados.
   - Compromiso para proseguir.
3. Uso de las consideraciones del riesgo para determinar el nivel de esfuerzo que se le debe dedicar a cada actividad dentro del cada ciclo de la espiral.
4. Uso de las consideraciones del riesgo para determinar el grado de detalle de cada artefacto producido en cada ciclo de la espiral.
5. Manejar las metas del ciclo de vida con tres hitos:
   - Objetivos del Ciclo de Vida (OCV).
   - Arquitectura del Ciclo de Vida (ACV).
   - Capacidad Operativa Inicial (COI).
6. Énfasis en las actividades y artefactos para el sistema y el ciclo de vida más que para el software y el desarrollo inicial.

A continuación se caracteriza al desarrollo en espiral enumerando algunas propiedades "invariantes" que cualquier proceso debe presentar y las razones por las cuales son esenciales. Para cada una, también se menciona un conjunto de "variantes", que demuestran un rango de definiciones de procesos en la familia de desarrollo en espiral. Se muestra además cómo se puede usar el modelo en espiral para un aprovechamiento incremental y más efectivo de los fondos. Una innovación importante y relativamente reciente al modelo en espiral ha sido la introducción de *hitos*.

Se adoptan muchos procesos que pueden parecer instancias del modelo en espiral, pero carecen de los invariantes esenciales. Cada invariante excluye uno o más de tales modelos de proceso, que se denominan "símil espiral peligroso".

### 2.3.1. Invariante Espiral 1

**Determinación Concurrente de Artefactos Clave (Concepto Operativo, Requerimientos, Planes, Diseño, Código)**

El Invariante Espiral 1, que se resume en la Figura 2.2 establece que un factor crítico para el éxito es determinar en forma concurrente una combinación compatible y factible de artefactos clave: el concepto operativo, los requerimientos del sistema y del software, los planes, la arquitectura y diseño del sistema y del software, y los componentes de código claves, incluyendo *COTS*, componentes reusados, prototipos, componentes críticos para el éxito, y algoritmos.

Los COTS (*Commercial-Off-The-Shelf*) son una clase especial de componentes de software, normalmente de granularidad gruesa, que presentan las siguientes características [BCK99]:

- Son vendidos o licenciados al público en general.
- Los mantiene y actualiza el propio vendedor, quien conserva los derechos de la propiedad intelectual.
- Están disponibles en forma de múltiples copias, todas idénticas entre sí.
- Su código no puede ser modificado por el usuario.

La cada vez mayor disponibilidad y uso de este tipo de componentes está impulsando notablemente la creación de un mercado global de componentes COTS. La

tecnología básica de componentes comienza a estar los suficientemente madura como para que numerosas empresas, e incluso gobiernos y ejércitos, la adopten en sus nuevos desarrollos y sistemas de información. Asimismo, están empezando a proliferar las empresas que venden con éxito componentes de software al mercado general.

---

## Resumen del Invariante 1

**Por qué invariante**

- Evita el compromiso secuencial prematuro con requerimientos del sistema, diseño, COTS, combinación de costo / tiempo / rendimiento.

**Ejemplo:**

- "tiempo de respuesta de un segundo"

**Variantes**

- 1a - monto relativo de cada artefacto desarrollado en cada ciclo.
- 1b - número de mini-ciclos concurrentes en cada ciclo.

**Modelos excluidos**

- Cascada secuencial incremental, con alto riesgo de violar las suposiciones del modelo en cascada.

---

Figura 2.2: Resumen del Invariante 1

¿Por qué es un invariante crítico para el éxito? Porque la determinación secuencial de los artefactos clave restringirá demasiado, y prematuramente, la posibilidad de desarrollar un sistema que satisfaga las condiciones de éxito esenciales para los interesados. Ejemplos de esto son los compromisos prematuros con ciertas plataformas de hardware, con combinaciones incompatibles de componentes COTS [GAO95], y con requerimientos cuyo cumplimiento no ha sido validado, como el mencionado requerimiento del tiempo de respuesta de un segundo, que se desarrolla a continuación.

Los variantes 1a y 1b indican que el interior del producto y del proceso de la actividad de ingeniería concurrente no es invariante. Para un sistema de interoperabilidad

de baja tecnología, los productos iniciales de la espiral serán intensivos en requerimientos. Para un sistema más integral, de alta tecnología, los productos iniciales de la espiral serán prototipos intensivos en código. Además, no hay un número fijo de mini-ciclos dentro de un ciclo espiral dado.

### Ejemplo: Tiempo de Respuesta de Un Segundo

La Figura 2.3 muestra un ejemplo de los tipos de problemas que surgen cuando se congela prematuramente los requerimientos de alto riesgo. A principios de los 80' una gran organización gubernamental contrató a TRW para desarrollar un sistema de información muy ambicioso. El sistema debería brindarle a más de mil usuarios, dispersos por todo un complejo edilicio, poderosas capacidades de consulta y análisis para una base de datos grande y dinámica.

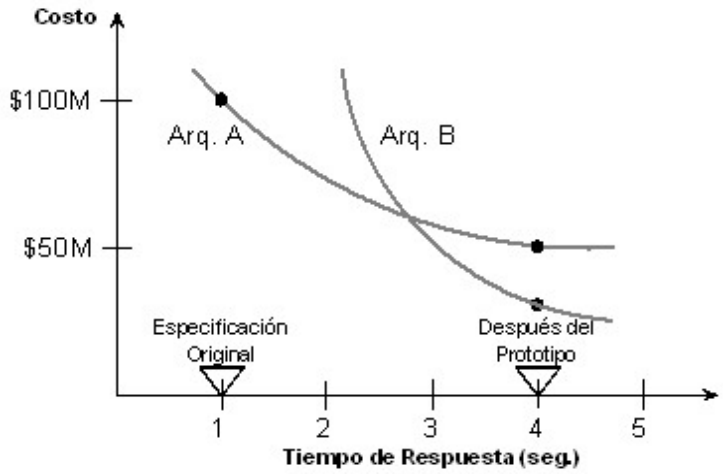


Figura 2.3: Dos diseños del sistema: Costo vs. Tiempo de Respuesta

TRW y el cliente especificaron el sistema usando un modelo clásico de desarrollo en cascada secuencial. Basándose principalmente en encuestas sobre las necesidades de los usuarios y en un análisis de desempeño de alto nivel demasiado simplificado, fijaron en el contrato un requerimiento para el tiempo de respuesta del sistema de menos de un segundo.

Dos mil páginas de requerimientos más tarde, los arquitectos de software encontraron que un rendimiento como el requerido sólo podía proveerse mediante la Arquitectura A, un diseño muy hecho a la medida que intentaba anticipar patro-

nes de consultas y guardar en caché copias de los datos de forma tal que los datos probables de cada usuario estarían dentro de los límites de un segundo. La arquitectura de hardware resultante tenía más de 25 súper mini-computadoras ocupadas guardando datos en caché de acuerdo a algoritmos cuyo rendimiento real desafiaba cualquier análisis. El alcance y la complejidad de la arquitectura hardware-software llevaron el costo estimado del sistema a cerca de $100 millones, debido principalmente al requerimiento del tiempo de respuesta de un segundo.

En vista de esta perspectiva poco atrayente, el cliente y el desarrollador decidieron crear un prototipo de la interfaz de usuario del sistema y de sus capacidades representativas, para someterlo a prueba. El resultado mostró que un tiempo de respuesta de cuatro segundos satisfacía a los usuarios el 90 por ciento de las veces. Un tiempo de respuesta de cuatro segundos podía lograrse con la Arquitectura B, bajando los costos de desarrollo a $30 millones [Boe00b]. Así, la especificación prematura de un tiempo de respuesta de un segundo introdujo el riesgo oculto de desarrollar un sistema demasiado costoso en tiempo y dinero.

**Símil espiral peligroso: Violación de las suposiciones del Modelo en Cascada.**

El Invariante 1 excluye un modelo que con frecuencia se clasifica como un proceso en espiral, pero que en realidad no lo es. Se trata de una secuencia de desarrollos en cascada incrementales con un alto riesgo de violar las suposiciones subyacentes del modelo en cascada. Estas suposiciones son:

1. Los requerimientos se conocen antes de la implementación.

2. Los requerimientos no tienen implicaciones de alto riesgo sin resolver, como riesgos debidos a elecciones de COTS, costo, planificación, rendimiento, seguridad, interfaces de usuario, e impactos organizacionales.

3. La naturaleza de los requerimientos no cambiará demasiado durante el desarrollo y la evolución.

4. Los requerimientos son compatibles con las expectativas de todos los interesados clave del sistema, incluyendo usuarios, clientes, desarrolladores, personal de mantenimiento, inversores.

5. Hay una buena comprensión de la arquitectura adecuada para implementar los requerimientos.

6. Hay suficiente tiempo como para proceder en forma secuencial.

Para que el modelo en cascada tenga éxito, un proyecto debe cumplir con estas suposiciones. Si todas son verdaderas, entonces es un riesgo del proyecto *no* especificar los requerimientos, y el modelo en cascada pasa a ser un caso especial del modelo en espiral conducido por el riesgo. Si cualquiera de estas suposiciones es falsa, entonces la especificación de un conjunto completo de requerimientos antes de la resolución de riesgos provocará incongruencias entre suposiciones y requerimientos, que pondrán en aprietos al proyecto.

La Suposición 1 –los requerimientos se conocen antes de la implementación– en general es falsa para sistemas interactivos nuevos, debido al síndrome IKIWISI. Cuando se les consulta sobre el esquema de pantalla que quieren para un nuevo sistema de soporte a las decisiones, los usuarios generalmente dirán "No sabría decirle, pero lo sabré cuando lo vea" (*I'll Know It When I See It* - IKIWISI). En esos casos es esencial un enfoque prototipo / requerimientos / arquitectura.

Los efectos de la no validez de las Suposiciones 2, 4, y 5 se ven perfectamente en el ejemplo de la Figura 2.3. El requerimiento de un tiempo de respuesta de un segundo no estaba resuelto y tenía un alto riesgo. Era compatible con las expectativas de los usuarios, pero no con las expectativas presupuestarias del cliente. Y la necesidad de una arquitectura muy cara para el cliente no estaba entendida de antemano.

Los efectos de la no validez de las Suposiciones 3 y 6 se pueden ver bien en los proyectos de comercio electrónico. En estos proyectos la volatilidad de la tecnología y del mercado es tal que las actualizaciones de requerimientos y trazas abrumarían al proyecto con sus costos. Más aún, la cantidad de tiempo calendario inicial que lleva formular un conjunto completo de requerimientos detallados que probablemente cambien varias veces con el tiempo no es una buena inversión del escaso tiempo disponible para desarrollar una capacidad operativa inicial.

### 2.3.2. Invariante Espiral 2

**Cada ciclo tiene Objetivos, Restricciones, Alternativas, Riesgos, Revisión, Compromiso para seguir**

El Invariante Espiral 2 identifica las actividades de cada cuadrante del diagrama espiral original que tienen que ejecutarse en cada ciclo espiral. Estas incluyen la consideración de los objetivos y restricciones de los interesados críticos; la elaboración y evaluación de alternativas de proyecto y proceso para lograr los objetivos sujetos a las restricciones; la identificación y resolución de los riesgos que acompañan a la elección de soluciones alternativas; y la revisión de los interesados y el compromiso a seguir en base a la satisfacción de sus objetivos y restricciones críticos. Si no se consideran todos estos factores, el proyecto puede comprometerse prematuramente con alternativas inaceptables para los interesados clave, o demasiado riesgosas. En la Figura 2.4 se resumen las características de este invariante.

El Invariante Espiral 2 no especifica la elección de una técnica particular de resolución de riesgos. Sin embargo, existen lineamientos de gestión de riesgos que sugieren, por ejemplo, las técnicas de resolución de riesgos del mejor candidato para las fuentes principales de riesgos del proyecto [Boe89]. Este invariante tampoco especifica niveles particulares de esfuerzo para las actividades ejecutadas durante cada ciclo. Los niveles deben estar balanceados entre los riesgos de aprender demasiado poco y los riesgos de derrochar tiempo y esfuerzo recogiendo información poco útil.

**Ejemplo: COTS sólo para Windows**

El hecho de ignorar el Invariante 2 puede llevar a malgastar mucho esfuerzo en elaborar una alternativa que pudo haberse demostrado como insatisfactoria mucho antes. Uno de los proyectos encarados por la biblioteca digital de la Universidad del Sur de California (USC) fue desarrollar un visualizador basado en la web para elementos de gran tamaño (por ejemplo, diarios o imágenes grandes). El prototipo inicial presentó una capacidad de visualización tremendamente poderosa y veloz, en base a un producto COTS llamado ER Mapper. La revisión del proyecto inicial aprobó la selección de este producto COTS, aún cuando sólo corría bien en plataformas Windows, y la Biblioteca tenía comunidades significativas de usuarios Macintosh y UNIX. Esta decisión se basó en las indicaciones iniciales de que pronto estarían disponibles

## Resumen del Invariante 2

**Por qué invariante**

- Evita el compromiso con alternativas inaceptables para los interesados, o demasiado riesgosas.
- Evita malgastar esfuerzos en elaborar alternativas insatisfactorias.

**Ejemplo:**

- "COTS sólo para Windows"

**Variantes**

- 2a - elección de técnicas de resolución de riesgos: prototipado, simulación, modelado, pruebas de referencia, etc.
- 2b - nivel de esfuerzo sobre cada actividad dentro de cada ciclo.

**Modelos excluidos**

- Fases secuenciales con interesados clave excluidos.

Figura 2.4: Resumen del Invariante 2

las versiones de ER Mapper para Mac y UNIX.

Sin embargo, posteriores investigaciones indicaron que pasaría mucho tiempo antes de que dichas capacidades para Mac y UNIX estuviesen disponibles. En una revisión posterior se descartó ER Mapper, a favor de un producto COTS menos poderoso pero completamente portable, Mr. SID, pero sólo después de mucho esfuerzo malgastado elaborando la solución con ER Mapper. Si en las primeras decisiones del proyecto se hubiese involucrado una comunidad representativa de usuarios Mac y UNIX, la elección de Mr. SID hubiese surgido antes, y se hubiese evitado el derroche de esfuerzo en elaborar la solución con ER Mapper.

**Símil espiral peligroso: Excluir interesados clave.**

Otro Símil espiral peligroso, excluido por el Invariante 2, consiste en organizar el proyecto en fases o ciclos secuenciales en los cuales los interesados clave quedan excluidos. Por ejemplo, excluir a los desarrolladores de la definición del sistema, excluir a los usuarios de la construcción del sistema, o excluir a los encargados del mantenimiento del sistema de la definición o de la construcción del mismo.

Aún cuando las fases que se muestran en la Figura 2.5 pueden asemejarse a ciclos espirales conducidos por riesgos, este símil espiral será peligroso porque su exclusión de interesados clave hará que sea difícil detectar riesgos críticos. Por ejemplo, excluir la participación del desarrollador en los primeros ciclos puede llevar a compromisos del proyecto basados en suposiciones equivocadas acerca de las capacidades del desarrollador [BR89].

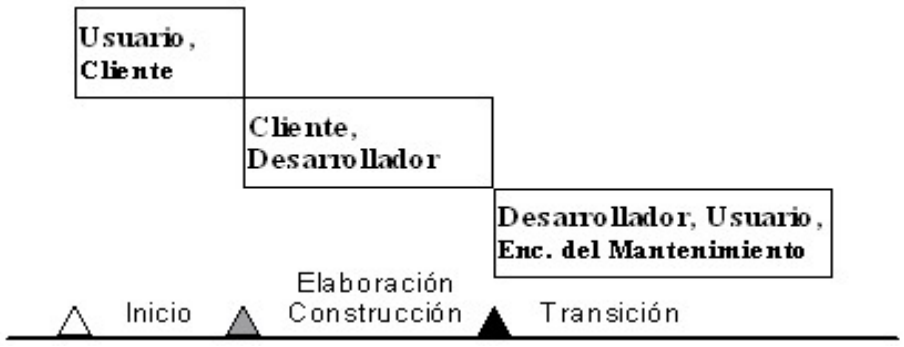


Figura 2.5: Modelos Excluidos: Fases Secuenciales sin Interesados Clave

### 2.3.3. Invariante Espiral 3

**Nivel de Esfuerzo Conducido por Consideraciones del Riesgo**

El Invariante Espiral 3, que se resume en la Figura 2.6, estipula el uso de consideraciones del riesgo para responder a las difíciles cuestiones de cuánto de una actividad dada es suficiente. ¿Cuánto de ingeniería del dominio es suficiente? ¿Cuánto de prototipado? ¿Cuánto de pruebas? ¿Cuánto de gestión de la configuración?, etc.

Si se grafica la exposición al riesgo de un proyecto como una función del tiempo consumido en la elaboración de prototipos, hay un punto en el cual la exposición al riesgo se minimiza. Utilizar más tiempo que ese es un desperdicio que lleva a una

## Resumen del Invariante 3

**Por qué invariante**

- Determina "cuánto es suficiente" de cada actividad: ingeniería del dominio, prototipado, pruebas, gestión de la configuración, etc.
- Evita la resolución tardía de los riesgos.

**Ejemplo:**

- Pruebas pre-entrega.

**Variantes**

- 3a - elección de los métodos usados para el seguimiento de las actividades: MBASE/WinWin, RUP de Rational, JAD, QFD, ESP, ...
- 3b - grado de detalle de los artefactos producidos en cada ciclo.

**Modelos excluidos**

- Desarrollo evolutivo o incremental insensible al riesgo.

Figura 2.6: Resumen del Invariante 3

entrada tardía y una menor penetración en el mercado. Utilizar menos tiempo que el indicado en la creación de prototipos lleva a un desarrollo prematuro con importantes demoras debidas a obstáculos imprevistos. Dado que los perfiles del riesgo varían de un proyecto a otro, esto implica que el nivel de minimización del riesgo del esfuerzo de prototipado variará de un proyecto a otro. La cantidad de esfuerzo dedicado a otras actividades también variará como una función del perfil del riesgo del proyecto.

Las variantes a considerar incluyen la elección de los métodos usados para el seguimiento de las actividades y el grado de detalle de los artefactos producidos en cada ciclo. Otra variante es una elección por parte de la organización de los métodos particulares para la evaluación y gestión de los riesgos.

**Ejemplo: pruebas pre-entrega**

La Figura 2.7 muestra cómo las consideraciones del riesgo pueden ayudar a determinar "cuántas pruebas son suficientes" antes de entregar un producto. Esto se determina sumando las dos principales fuentes de Exposición al Riesgo, ER = Probabilidad (Pérdida) * Tamaño (Pérdida), incurridas por las dos fuentes de pérdidas: pérdidas en la rentabilidad debidas a defectos en el producto, y pérdidas en la rentabilidad debidas a demoras en la captura de una porción del mercado. Cuanto más pruebas se efectúen, disminuye el riesgo de entregar productos con defectos. Sin embargo, cuanto más tiempo se consume probando, aumentan tanto la probabilidad de pérdida debido a competidores que entran al mercado como el monto de las pérdidas debidas a la disminución de la rentabilidad en la porción de mercado restante.

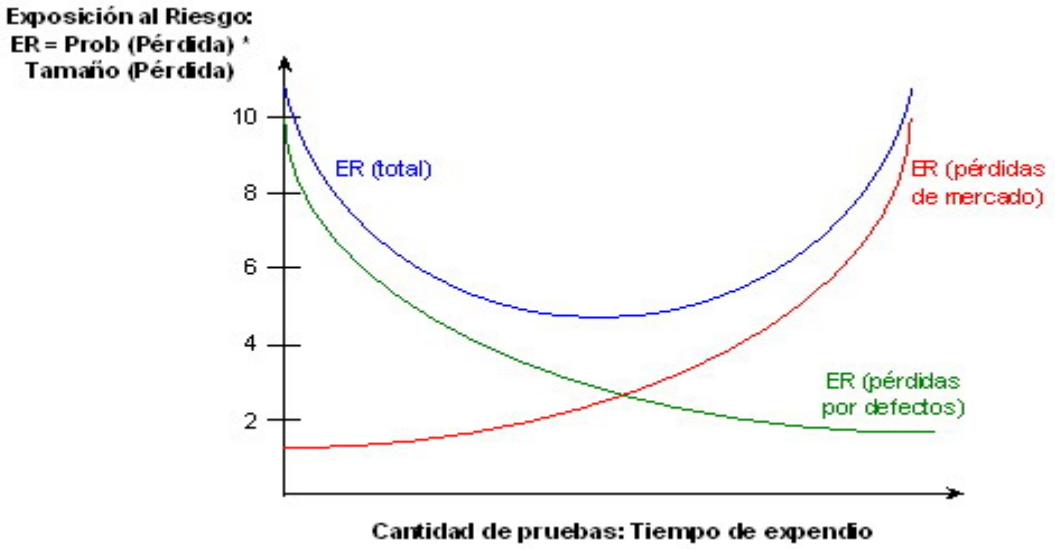


Figura 2.7: Exposición al Riesgo en las Pruebas Pre-entrega

Como se muestra en la Figura 2.7, la suma de estas exposiciones al riesgo alcanza un mínimo en algún nivel intermedio de las pruebas. La ubicación en el tiempo de este punto de mínimo riesgo variará de acuerdo al tipo de organización. Por ejemplo, será considerablemente menor en una compañía comercial cualquiera que en un producto cuya seguridad es crítica, como una planta de energía nuclear. El cálculo de la exposición al riesgo también requiere que una organización acumule una buena cantidad de experiencia en las probabilidades y montos de las pérdidas como funciones de la duración de las pruebas y de las demoras en la entrada al mercado.

**Símil Espiral Peligroso: Insensibilidad al Riesgo.**

Los modelos peligrosos que se parecen al Espiral y que son excluidos por el Invariante 3 son:

- Desarrollo evolutivo insensible al riesgo (por ejemplo, descuidar riesgos debidos a la escalabilidad).
- Desarrollo incremental insensible al riesgo.
- Planes espirales impecables sin compromiso de manejar los riesgos identificados.

### 2.3.4. Invariante Espiral 4

**Grado de Detalle Conducido por Consideraciones del Riesgo**

El Invariante Espiral 4 (Figura 2.8) es la contraparte a nivel producto del Invariante 3: que las consideraciones del riesgo determinan el grado de detalle de los productos como así también de los procesos. Esto significa, por ejemplo, que el ideal tradicional de una especificación de requerimientos completa, consistente, rastreable, y comprobable no es una buena idea para ciertos componentes del producto, como una interfaz gráfica de usuario (IGU) o una interfaz de un componente COTS. Aquí el riesgo de especificar en forma precisa las distribuciones de pantalla antes del desarrollo implica una alta probabilidad de bloquear una interfaz de usuario complicada dentro del contrato de desarrollo, mientras que el riesgo de no especificar distribuciones de pantalla es bajo, dada la disponibilidad general de herramientas flexibles de diseño de IGUs. Incluso aspirar a una consistencia y comprobabilidad completa puede ser riesgoso, porque crea presión para especificar prematuramente decisiones que podría ser mejor diferir (por ejemplo, la forma y contenido de los informes de excepciones). Sin embargo, algunos patrones de riesgo hacen que sea muy importante tener especificaciones precisas, como el riesgo de incompatibilidades críticas de interfaces entre componentes de software y hardware, o entre el software de un contratista primario y el de un subcontratista.

Estos lineamientos muestran cuándo es riesgoso especificar de más o de menos las características del software:

- Si es riesgoso *no* especificar precisamente, entonces *especificar* (por ejemplo, interfaz hardware-software, interfaz primario-subcontratista, etc.).

- Si es riesgoso especificar precisamente, entonces *no especificar* (por ejemplo, distribuciones de IGUs, comportamiento de los COTS, etc.).

Las variantes espirales referidas al Invariante 4 son las elecciones de representación para los artefactos producidos.

---

## Resumen del Invariante 4

**Por qué invariante**

- Determina "cuánto es suficiente" de cada artefacto (Requerimientos, Diseño, Código, Planes) en cada ciclo.
- Evita la resolución tardía de los riesgos.

**Ejemplo:**

- Riesgo de la Especificación Precisa.

**Variantes**

- Elección de representaciones de artefactos (SA/SD, UML, MBASE, especificaciones formales, lenguajes de programación, etc.)

**Modelos excluidos**

- Especificación de requerimientos completa, consistente, rastreable, y comprobable para sistemas que involucran niveles significativos de IGUs, COTS, o decisiones diferidas.

---

Figura 2.8: Resumen del Invariante 4

### Ejemplo: Riesgo de Especificación Precisa.

La especificación de un editor de textos requiere que cada operación esté disponible mediante un botón en la ventana. En consecuencia, el espacio disponible para ver y editar el texto queda demasiado pequeño. El desarrollador no puede mover algunas operaciones a los menús porque la distribución de la IGU se especificó precisamente en

una etapa anterior. (Por supuesto, si se les da demasiada libertad a los programadores, pueden desarrollar IGUs muy malas. La revisión de los interesados es esencial para evitar esos problemas.)

### 2.3.5. Invariante Espiral 5

#### Uso de Hitos de Anclaje: OCV, ACV, COI

Una dificultad importante del modelo en espiral original fue su carencia de hitos intermedios que sirvan como puntos de compromiso y de control del progreso [FMC96]. Esta carencia fue remediada mediante la inclusión de un conjunto de hitos de anclaje: Objetivos del Ciclo de Vida (OCV), Arquitectura del Ciclo de Vida (ACV), y Capacidad Operativa Inicial (COI) [Boe96]. Estos se pueden describir como puntos de compromiso de los interesados en el ciclo de vida del software: OCV es el compromiso de los interesados para respaldar la arquitectura; ACV es el compromiso de los interesados para mantener el ciclo de vida completo; y COI es el compromiso de los interesados para sostener las operaciones. El invariante 5 se resume en la Figura 2.9.

Los hitos de anclaje fueron definidos en un par de talleres del Centro de Afiliados de la Ingeniería del Software de la Universidad del Sur de California, y como tales representan un esfuerzo conjunto de participantes tanto de la industria como del gobierno [CB95]. Uno de los afiliados, *Rational, Inc.*, había estado definiendo las fases de su Proceso Unificado, y adoptó los hitos de anclaje como la entrada a cada una de esas fases.

Los dos primeros puntos de anclaje son los Objetivos del Ciclo de Vida (OCV), y la Arquitectura del Ciclo de Vida (ACV). En cada uno de ellos, los interesados clave revisan seis artefactos: descripción del concepto operativo, resultados de los prototipos, descripción de los requerimientos, descripción de la arquitectura, plan del ciclo de vida, y análisis de factibilidad (para más detalles, ver la Sección 2.4.1).

El análisis de factibilidad cubre la pregunta de aprobación clave: “si se construye este producto usando la arquitectura y procesos especificados, ¿soportará el concepto operativo, cumplirá los resultados de los prototipos, satisfará los requerimientos, y terminará dentro de lo previsto en cuanto a tiempo y dinero?”. Si la respuesta es no, el paquete deberá trabajarse nuevamente.

## Resumen del Invariante 5

**Por qué invariante**

- Evita la parálisis del análisis, expectativas poco realistas, atraso de requerimientos, deficiencias e incompatibilidades debidas a COTS, arquitecturas insostenibles o sin rumbo preciso, recortes traumáticos, sistemas inútiles.

**Variantes**

- 5a - número de ciclos espirales o incrementos entre puntos de anclaje.
- 5b - fusión de puntos de anclaje de acuerdo a la situación.

**Modelos excluidos**

- Desarrollo evolutivo o incremental sin arquitectura del ciclo de vida.

Figura 2.9: Resumen del Invariante 5

El objetivo de la revisión de los OCV es asegurar que al menos una elección de arquitectura es viable desde la perspectiva del negocio. El objetivo de la revisión de la ACV es el compromiso con una única definición detallada de los artefactos de la revisión. El proyecto debe haber eliminado todos los riesgos significativos, o establecido un plan aceptable de manejo de riesgos. El hito de la ACV es particularmente importante, porque su criterio de aprobación les permite a los interesados sostener proyectos que intentan proseguir con un desarrollo evolutivo o incremental sin una arquitectura del ciclo de vida.

El hito OCV es el equivalente a comprometerse, y el hito ACV es el equivalente a casarse. Como en la vida, casarse demasiado pronto con una arquitectura provocará arrepentimientos al poco tiempo. El tercer hito de anclaje, la Capacidad Operativa Inicial (COI), constituye un compromiso aún mayor: es el equivalente a tener el primer hijo.

Las variantes apropiadas incluyen el número de ciclos espirales de incrementos de desarrollo entre los puntos de anclaje. En algunos casos, estos hitos pueden fusionarse. En particular, un proyecto que decide usar un lenguaje de cuarta generación maduro

y apropiadamente escalable ya habrá determinado su elección de la arquitectura del ciclo de vida en el hito OCV, permitiendo que OCV y ACV se fusionen.

**Modelo espiral y compromiso incremental.**

Un aspecto valioso de la aplicación original del modelo espiral al Sistema de Productividad del Software TRW fue su capacidad de soportar el compromiso incremental de los recursos corporativos a la exploración, definición, y desarrollo del sistema, en lugar de solicitar un gran desembolso para el proyecto antes de que sus perspectivas de éxito estén bien comprendidas [Boe88]. Estas decisiones se codifican con los lineamientos específicos de OCV y ACV.

### 2.3.6. Invariante Espiral 6

**Énfasis en las Actividades y Artefactos del Sistema y del Ciclo de Vida**

El Invariante Espiral 6, resumido en la Figura 2.10, enfatiza que el desarrollo en espiral de sistemas de software necesita enfocarse no sólo en los aspectos de la construcción del software, sino también en el sistema completo y en las incumbencias del ciclo de vida. Los desarrolladores de software son tienden a caer con facilidad en la trampa muchas veces citada: "si nuestra mejor herramienta es un martillo, el mundo que vemos es una colección de clavos" [Boe00a]. Escribir código puede ser un fuerte del desarrollador, pero tiene la misma importancia en el proyecto que la que tienen los clavos en una casa.

El énfasis del modelo en espiral en usar los objetivos de los interesados para conducir las soluciones del sistema, y en los hitos de anclaje del ciclo de vida, guía a los proyectos a concentrarse en las incumbencias del sistema y del ciclo de vida. El uso de las consideraciones del riesgo que hace el modelo para encontrar soluciones posibilita el ajuste de cada ciclo espiral a cualquier combinación de software y hardware, elección de capacidades, o grado de "productización" que sean apropiados.

**Ejemplo: "Procesamiento de órdenes".**

Un buen ejemplo es el sistema de procesamiento de órdenes de *Scientific American* esquematizado en la Figura 2.11. El personal de informática buscó parte del problema

## Resumen del Invariante 6

**Por qué invariante**

- Evita la sub-optimización prematura de hardware, software, o consideraciones de desarrollo.

**Ejemplo:**

- Procesamiento de órdenes.

**Variantes**

- 6a - cantidad relativa de hardware y software determinada en cada ciclo.
- 6b - cantidad relativa de capacidad en cada incremento del ciclo de vida.
- 6c - grado de "productización" (alfa, beta, etc.) de cada incremento del ciclo de vida.

**Modelos excluidos**

- Métodos orientados a objetos puramente lógicos (porque son insensibles a riesgos operativos, de costo y de rendimiento).

Figura 2.10: Resumen del Invariante 6

con una solución de software (su "clavo"), y lo clavó con su martillo de software, dejando a la *Scientific American* peor de lo que estaba.

Los objetivos de la *Scientific American* eran reducir los costos, errores, y demoras de su sistema de procesamiento de suscripciones. En lugar de analizar las fuentes de estos problemas, la compañía de software se concentró en la parte del problema que tenía una solución de software. El resultado fue un sistema computacional de procesamiento por lotes cuyas largas demoras provocaron un esfuerzo adicional sobre la porción del sistema que había sido la principal fuente de costos, demoras, y errores en primera instancia. Como se ve en el gráfico, el resultado del negocio fue un sistema nuevo con más errores, demoras más largas, costos más elevados, y trabajo menos

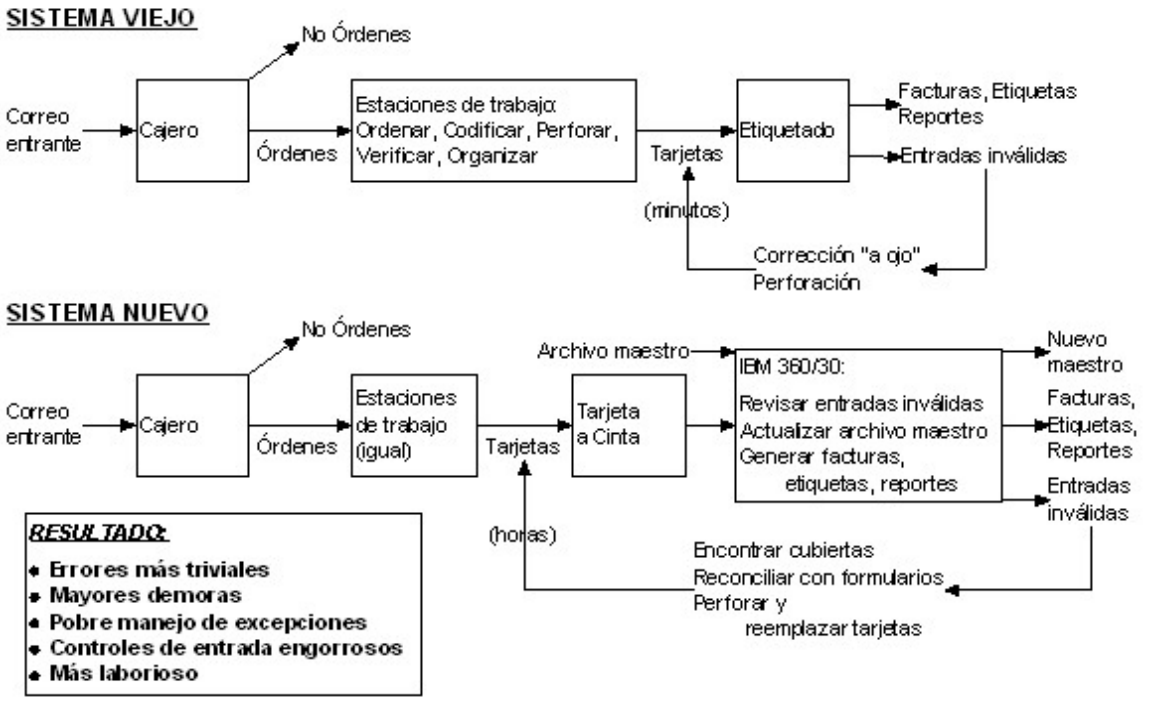


Figura 2.11: Procesamiento de Órdenes de *Scientific American*

atractivo que su predecesor [Boe81].

Este tipo de resultados se hubiese dado incluso si el software que automatizaba las funciones de la máquina de etiquetar se hubiese desarrollado en un enfoque cíclico conducido por el riesgo. Sin embargo, su paquete correspondiente al hito de los Objetivos del Ciclo de Vida no hubiese pasado su revisión de factibilidad, porque no tenía un caso de negocio a nivel sistema que demostrara que el desarrollo del software llevaría a la reducción deseada de costos, errores, y demoras. De haberse hecho un análisis minucioso del caso de negocio, se hubiese detectado la necesidad de una reingeniería en los procesos de oficina como así también de automatizar los procesos manuales de etiquetación. Más aún, como muestran métodos recientes como el Enfoque de Logro de Beneficios de DMR, se podría haber usado el caso de negocio para monitorear el cumplimiento real de los beneficios esperados, y para aplicar acciones correctivas a la reingeniería del proceso de negocio o a las porciones de ingeniería de software de la solución (o a ambas) que se consideren apropiadas [Tho98].

**Símil espiral peligroso: Diseños Orientados a Objetos puramente lógicos.**

Los modelos excluidos por el Invariante 6 incluyen los métodos de análisis y diseño orientado a objetos más conocidos, los que usualmente se presentan como ejercicios lógicos abstractos independientes del desempeño del sistema o de incumbencias económicas. Por ejemplo, una reciente inspección de 16 libros sobre análisis y diseño orientado a objetos reveló que en sólo seis de ellos la palabra "desempeño" ("*performance*") aparecía en su índice, y en sólo dos aparecía la palabra "costo" [Boe00b].

## 2.4. Hitos de Anclaje

Los hitos de anclaje que surgen del invariante 5 son [Boe00a]:

- Objetivos del Ciclo de Vida (OCV)
- Arquitectura del Ciclo de Vida (ACV)
- Capacidad Operativa Inicial (COI)

Estos hitos son agregados relativamente nuevos al modelo de desarrollo en espiral [Boe96]. A continuación se verán más en profundidad.

### 2.4.1. Descripciones Detalladas

El Cuadro 2.1 muestra las características principales de los hitos OCV y ACV. A diferencia de la mayoría de los hitos de software actuales:

- Su foco no se encuentra en instantáneas de los requerimientos o en soluciones puntuales de la arquitectura, sino en requerimientos y especificaciones arquitectónicas que se anticipan y adaptan a la evolución del sistema. Esta es la razón para llamarlos Objetivos e Hitos Arquitectónicos del "Ciclo de Vida".
- Los elementos pueden ser especificaciones o programas ejecutables con datos (por ejemplo, prototipos, productos COTS).
- El Análisis de Factibilidad es un elemento esencial, y no un agregado opcional.

- La concurrencia de los interesados sobre los elementos del hito es esencial. Esto permite una asociación mutua de los interesados con los planes y especificaciones, y permite un enfoque de equipo en colaboración ante problemas imprevistos, en lugar del enfoque adversario que caracteriza a la mayoría de los modelos por contrato.

Estas características explican por qué OCV y ACV son críticos para el éxito de los proyectos, y por qué pueden funcionar exitosamente como puntos de anclaje en muchos tipos de desarrollos de software.

Una característica clave del hito OCV es la necesidad de que el Análisis de Factibilidad demuestre un caso de negocio viable para el sistema propuesto. Este caso de negocio no sólo debería mantenerse actualizado, sino que también debería usarse como base para verificar que realmente se alcanzarán los beneficios esperados, como se discutió en el ejemplo del "Procesamiento de Órdenes" del Invariante 6.

Una característica que diferencia al hito ACV del OCV es la necesidad de tener resueltos todos los riesgos principales del sistema, o al menos cubiertos por algún elemento del plan de gestión de riesgos del sistema. Para sistemas grandes, pasar el hito ACV es el punto de ascenso significativo de nivel del personal y de compromiso de recursos. La continuación dentro de esta etapa sin haber tratado los riesgos principales ha producido desastres en muchos proyectos grandes.

La Capacidad Operativa Inicial (COI) es lo primero que verán los usuarios de un sistema en funcionamiento, por lo que tener cosas mal hechas en el COI puede producir serias consecuencias. Recibir a los usuarios con un nuevo sistema que tiene software mal acoplado, una preparación pobre de la situación o de los usuarios, ha sido causa frecuente de alienación de los usuarios y de falla del proyecto.

Los elementos clave del hito COI son

- *Preparación del software*, que incluye software operativo y de soporte, con comentarios y documentación apropiados; preparación o conversión de datos; las licencias y derechos necesarios para COTS y software reusado, y las debidas pruebas finales de operatividad.

- *Preparación de la situación*, que incluye facilidades, equipamiento, suministros, y acuerdos de soporte con los vendedores de COTS.

| Elemento Hito | Objetivos del Ciclo de Vida (OCV) | Arquitectura del Ciclo de Vida (ACV) |
|---|---|---|
| **Definición del Concepto Operativo** | Objetivos y alcance de nivel superior del sistema<br>• Límite del sistema<br>• Parámetros y suposiciones del entorno<br>• Parámetros de evolución<br>Concepto operativo<br>• Escenarios y parámetros de operaciones y mantenimiento<br>• Responsabilidades del ciclo de vida de la organización (interesados) | Elaboración de los objetivos del sistema y del alcance de los incrementos.<br>Elaboración del concepto operativo por incremento. |
| **Prototipo(s) del Sistema** | Ejercitar escenarios de uso claves.<br>Resolver riesgos críticos. | Ejercitar rango de escenarios de uso.<br>Resolver los riesgos más destacados. |
| **Definición de Requerimientos del Sistema** | Funciones, interfaces, niveles de atributos de calidad, incluyendo:<br>• Vectores y prioridades de crecimiento.<br>• Prototipos.<br>Concurrencia de los interesados en las bases | Elaboración de funciones, interfaces, atributos de calidad, y prototipos por incremento:<br>• Identificación de ítems a determinar.<br>Concurrencia de los interesados en sus intereses de prioridades. |
| **Definición de la Arquitectura del Software y del Sistema** | Definición de alto nivel de al menos una arquitectura viable.<br>• Elementos y relaciones físicas y lógicas.<br>• Elecciones de COTS y elementos de software reusables.<br>Identificación de opciones inviables. | Elección de la arquitectura y elaboración por incremento<br>• Componentes, conectores, configuraciones, restricciones, físicos y lógicos.<br>• Elecciones de COTS y elementos reusables.<br>• Elecciones de arquitectura del dominio y del estilo arquitectónico.<br>Parámetros de evolución arquitectónica |
| **Definición del Plan del Ciclo de Vida** | Identificación de los interesados del ciclo de vida.<br>• Usuarios, clientes, desarrolladores, personal de mantenimiento, operadores, público en general, otros.<br>Identificación del modelo de proceso del ciclo de vida.<br>• Etapas e incrementos de alto nivel.<br>PQCQDCC* de alto nivel por etapa. | Elaboración de PQCQDCC* para la Capacidad Operativa Inicial (COI).<br>• Elaboración e identificación parcial de los ítems clave a determinar en incrementos posteriores. |
| **Análisis de Factibilidad** | Asegurar la consistencia entre los elementos anteriores.<br>• Mediante análisis, medición, prototipado, simulación, etc.<br>• Análisis del caso de negocio para requerimientos, arquitectura viable. | Asegurar la consistencia entre los elementos anteriores.<br>Todos los riesgos principales resueltos o cubiertos por el plan de gestión de riesgos. |

* PQCQDCC: Por qué, Qué, Cuándo, Quién, Dónde, Cómo, Cuánto

Cuadro 2.1: Puntos de Anclaje Espiral

- *Preparación de usuarios, operadores, y personal de mantenimiento*, que incluye selección, conformación de equipos, entrenamiento y otras capacitaciones para familiarizarse, usar, operar, o mantener el sistema.

Como en el ejemplo de las Pruebas Pre-Entrega del Invariante 3, el hito COI es dirigido por los riesgos con respecto a los objetivos del sistema determinados en los hitos OCV y ACV. Así, por ejemplo, estos objetivos dirigen la confrontación entre la fecha del COI y la calidad del producto. Estos diferirán marcadamente entre sistemas tales como el Software para una Plataforma Espacial y un producto de software comercial. La diferencia entre estos dos casos se está haciendo cada vez menor, a medida que los vendedores y usuarios comerciales aprecian cada vez más los riesgos de mercado que los productos defectuosos significan [CS95b].

## 2.5. Extensiones y Mejoras al Modelo en Espiral

El modelo en espiral comienza cada ciclo con el próximo nivel de elaboración de los objetivos, restricciones y alternativas del potencial sistema. Una dificultad importante al aplicar el modelo en espiral original ha sido la carencia de una guía explícita para determinar esos objetivos, restricciones y alternativas. Para resolver ese problema, Boehm y Bose presentaron en [BB94] una extensión del modelo en espiral, a la que llamaron *Next Generation Process Model* (NGPM), que usa el enfoque de la *Teoría W* (*win-win*) también propuesta por Boehm [BR89], para determinar los objetivos, restricciones y alternativas del próximo nivel del sistema. A continuación se introducen brevemente los principios de la Teoría W, y luego se presenta la mencionada extensión al modelo espiral original, NGPM.

### 2.5.1. Teoría W de Gestión de Proyectos de Software

La consigna principal de la Teoría W es *hacer que todos los involucrados ganen* [BR89]. Intenta entonces resolver el problema principal del director de un proyecto de software, que es satisfacer simultáneamente las necesidades de varios integrantes: los usuarios, los clientes, el equipo de desarrollo, el equipo de mantenimiento, los altos mandos, etc. Pero cuando todas esas necesidades se toman en conjunto, pueden aparecer conflictos fundamentales, como por ejemplo muchas funciones en el producto

(requeridas por los usuarios) contra un bajo presupuesto (requerido por el cliente).

En este contexto, el desafío es crear situaciones donde todos ganen (*win-win*), poniendo especial atención a los intereses y expectativas de todas las partes involucradas. Si esto no se logra, pueden aparecer situaciones donde una parte gane pero la otra pierda (*win-lose*), o lo que es peor, situaciones donde todas las partes pierdan (*lose-lose*). Por ejemplo, la construcción rápida y descuidada de un producto puede ser una ganancia para los desarrolladores y clientes por su bajo costo y su rápida entrega (*win*), pero una situación de pérdida para las personas que lo deben usar o mantener (*lose*). Y hay muchos ejemplos de situaciones que generalmente terminan en pérdidas para todos los participantes (*lose-lose*): establecer un cronograma poco realista, contratar personal incompatible, planificar pobremente, etc.

Los pasos propuestos por los autores de la Teoría W son:

1. Establecer un conjunto de precondiciones *win-win*.
    *a*) Comprender cómo quieren ganar las personas.
    *b*) Establecer expectativas razonables.
    *c*) Hacer corresponder las tareas de las personas con sus condiciones de ganancia.
    *d*) Proveer un ambiente comprensivo.
2. Estructurar un proceso de software *win-win*.
    *a*) Establecer un plan de proceso realista.
    *b*) Usar el plan para controlar el proyecto.
    *c*) Identificar y manejar los riesgos *win-lose* o *lose-lose*.
    *d*) Mantener involucradas a las personas.
3. Estructurar un producto de software *win-win*.
    *a*) Hacer corresponder el producto con las condiciones de ganancia de usuarios y personal de mantenimiento.

### 2.5.2. NGPM (*Next Generation Process Model*)

En base a la Teoría W [BR89] presentada en la sección anterior, [BB94] presentan las extensiones al modelo en espiral, que se ilustra en la Figura 2.12. Agregan dos sectores adicionales al principio de cada ciclo de la espiral, "Identificar los interesados del próximo nivel" e "Identificar las condiciones de ganancia de los interesados", y una porción en el tercer sector, "Reconciliar las condiciones de ganancia". Estos agregados proveen las bases de colaboración para el modelo. Además llenan un vacío del modelo espiral original, que es proveer los medios para responder a las preguntas "¿De dónde vienen los objetivos y restricciones del próximo nivel?" y "¿Cómo se sabe que son los correctos?".

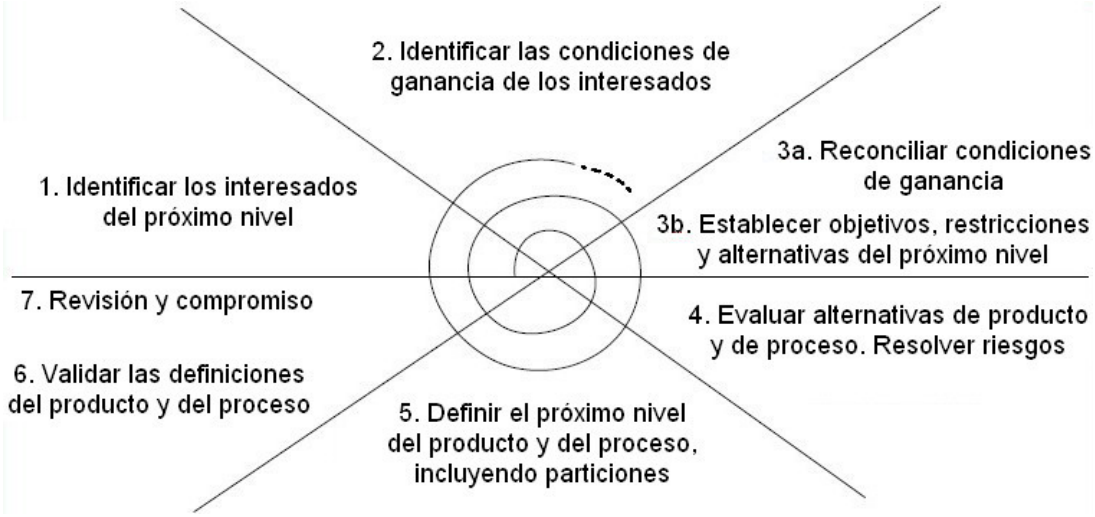


Figura 2.12: Modelo en Espiral con las extensiones de NGPM

El modelo espiral refinado también se refiere explícitamente a la necesidad de efectuar de manera concurrente el análisis, la resolución de riesgos, la definición y la elaboración tanto del producto como del proceso de software. En particular, el proceso de nueve pasos de la Teoría W se traduce en las siguientes extensiones del modelo en espiral:

- Determinar objetivos. Identificar los interesados del ciclo de vida del sistema y sus condiciones de ganancia. Establecer límites del sistema e interfaces externas iniciales.

- Determinar restricciones. Determinar las condiciones bajo las cuales el sistema produciría resultados indeseables para algunos interesados.
- Identificar y evaluar alternativas. Solicitar sugerencias a los interesados. Evaluarlas con respecto a las condiciones de ganancia de los interesados. Sintetizar y negociar las alternativas de ganancia candidatas. Analizar, evaluar y resolver los riesgos de pérdidas.
- Registrar compromisos, y áreas que deben ser más flexibles, en el registro del diseño del proyecto y en los planes del ciclo de vida.
- Recorrer la espiral. Elaborar condiciones de ganancia, filtrar alternativas, resolver riesgos, acumular compromisos apropiados, y desarrollar y ejecutar los planes.

Un proyecto que sigue el NGPM involucra a los interesados (usuarios, clientes, desarrolladores, encargados del mantenimiento y del diseño de interfaces, etc.) para que colaboren dentro de la estructura del modelo. Los autores desarrollaron un sistema de soporte para el modelo, y en [BB94] describen alguna experiencias en la aplicación del mismo.

Por otro lado, en [B+98] se presenta un estudio de un caso en el que 15 equipos usaron el modelo en espiral WinWin para prototipar, planificar, especificar y construir aplicaciones multimedia para un Sistema Integrado de Biblioteca para la Universidad del Sur de California.

## 2.6. Un modelo de valoración de riesgos para proyectos de software evolutivos

A pesar del progreso en los métodos formales, el prototipado, y los procesos de software evolutivos, la valoración de riesgos sigue siendo un tema abierto que depende de la experiencia humana. Algunos procesos de desarrollo de software, como el modelo en espiral, tienen una debilidad común: la valoración de riesgos [Nog00b]. En el dominio de la evolución del software, la valoración de riesgos no se ha abordado como parte del modelo. En las diferentes mejoras y extensiones, el modelo no incluye pasos de valoración de riesgos, y por lo tanto la gestión de riesgos sigue siendo una

actividad que depende de los seres humanos, y que requiere de experiencia. En la evaluación del modelo en espiral, una de las dificultades que mencionó Boehm fue: *"Al depender de la experiencia en la valoración de riesgos, el modelo en espiral confía en gran medida en la habilidad de los desarrolladores de software para identificar y manejar las fuentes de los riesgos del proyecto."* [Boe88]

Muchas investigaciones han abordado el problema de la valoración de riesgos siguiendo lineamientos, listas de comprobación, taxonomías de factores de riesgo, y algunas métricas. Todos estos métodos funcionan bien si a) los aplica una persona educada en valoración de riesgos, y b) esa persona tiene suficiente experiencia. La debilidad de todas las prácticas actuales de valoración de riesgos es la dependencia humana. Como corolario, la valoración de riesgos podría no ser consistente, debido a que diferentes expertos podrían llegar a conclusiones distintas del mismo escenario.

Nogueira [Nog00b] presenta un modelo para valorar automáticamente los riesgos y la duración de los proyectos de software, en base a indicadores objetivos que pueden medirse en las primeras etapas del proceso. El modelo se ha diseñado para tener en cuenta características importantes de los procesos de software evolutivos, como la complejidad de los requerimientos, su volatilidad, y la eficiencia organizacional. El modelo formal basado en estos tres indicadores estima la duración y riesgo de procesos de software evolutivo. El enfoque soporta automatización de la valoración de riesgos y métodos de estimación temprana.

### 2.6.1. El problema

A medida que el alcance y la complejidad de las aplicaciones informáticas han ido creciendo, el costo del desarrollo de software se ha transformado en el mayor gasto de los sistemas basados en computadoras [Boe81, Kar96]. Los excesos en los tiempos y costos previstos son trágicamente comunes, como lo muestran varias investigaciones tanto en la industria privada como en el ámbito gubernamental [Boe81, Luq89, Jon94]. A pesar de las mejoras en herramientas y metodologías, hay poca evidencia de éxito en la mejora del proceso de pasar del concepto al producto, y se ha progresado muy poco en la gestión de proyectos de desarrollo de software [Hal98]. La valoración de riesgos sigue siendo un problema desestructurado que depende de la experiencia humana. Todavía se carece de formas sistemáticas de identificar, comunicar, y resolver incertidumbres técnicas [HH96].

Es necesario entonces construir un modelo de valoración de riesgos en base a parámetros objetivamente medibles que se puedan recolectar y analizar automáticamente. Resolver el problema de la valoración de riesgos con indicadores medidos en las etapas tempranas constituiría un gran beneficio a la ingeniería de software, ya que en estas etapas se pueden realizar cambios con el menor impacto en el presupuesto.

La fase de requerimientos es crucial para valorar riesgos porque:

1. involucra una gran cantidad de interacción y comunicación humana que puede ser malinterpretada y dar lugar a errores;

2. los errores introducidos en esta fase son muy difíciles de corregir si se descubren tarde;

3. la existencia de herramientas de generación de software puede disminuir los errores en el proceso de desarrollo si los requerimientos son correctos; y

4. los requerimientos evolucionan introduciendo cambios y mantenimiento a lo largo de todo el ciclo de vida.

### 2.6.2. El Modelo Propuesto para Riesgos del Proyecto

El enfoque de Nogueira [Nog00b] se basa en métricas que se pueden recolectar automáticamente de la base de datos de la ingeniería, prácticamente desde el principio del desarrollo. Los indicadores que se usan son Volatilidad de Requerimientos *(VR)*, Complejidad *(CX)*, y Eficiencia *(EF)*.

*Volatilidad de Requerimientos (VR):* VR es una medida de tres características de los requerimientos:

- la Tasa de Nacimientos *(TN)*, que es el porcentaje de nuevos requerimientos incorporados en cada ciclo;

- la Tasa de Muertes *(TM)*, que es el porcentaje de requerimientos descartados en cada ciclo; y

- la Tasa de Cambio *(TC)*, que se define como el porcentaje de requerimientos que cambiaron respecto a la versión anterior. Un cambio en un requerimiento se modela como el nacimiento de un nuevo requerimiento y la muerte de otro, por lo que TC se incluye en los valores medidos de TN y TM.

Entonces, VR se calcula como sigue: $VR = TN + TM$.

*Complejidad (CX):* la complejidad de los requerimientos se mide desde una especificación formal. Una representación de requerimientos que soporta el prototipado asistido por computadora, como PSDL [Luq96], es útil en el contexto del prototipado evolutivo. Se define una métrica de complejidad llamada Complejidad de Granularidad Gruesa *(CGG)* que se calcula como: $CGG = O + D + T$, donde $O$ es el número de operadores atómicos (funciones o máquinas de estados), $D$ es el número de flujos de datos (conexiones de datos entre operadores), y $T$ es el número de tipos de datos abstractos requeridos para el sistema. Los operadores y los flujos de datos son los componentes de un diagrama de flujo de datos. Esta es una medida de la complejidad de la arquitectura del prototipo, similar en espíritu a los puntos de función, pero más adecuados para modelar sistemas embebidos y de tiempo real. La medida también se puede aplicar a otras notaciones de modelado que representen módulos, conexiones de datos, y tipos de datos abstractos o clases. Existe una fuerte correlación entre la complejidad medida en CGG y el tamaño de las especificaciones PSDL.

*Eficiencia (EF):* la eficiencia de la organización se mide usando una observación directa del uso del tiempo. EF se calcula como una razón entre el tiempo dedicado a la labor directa y el tiempo ocioso: *EF = Tiempo de Labor Directa / Tiempo Ocioso.* Se descubrió que este valor de fácil obtención era un buen discriminante entre una productividad alta del equipo y una baja en un conjunto de proyectos de software simulados [Nog00a].

Los autores validaron y calibraron el modelo con una serie de proyectos de software simulados, usando Vit Project. Los parámetros de entrada para los escenarios simulados fueron VR, EF, y CX, y la salida observada fue el tiempo de desarrollo. Dado que el modelo propuesto usa parámetros recogidos durante las etapas tempranas y que Vit Project requiere un desglose completo de la estructura para el proyecto, lo que sólo puede hacerse en las fases tardías, hubo una brecha de tiempo considerable entre las dos medidas. De todos modos, los resultados fueron suficientes para propósitos de calibración y validación del modelo.

Los resultados de la simulación se analizaron estadísticamente, y surgió que la distribución de probabilidad que mejor se ajustaba a todos las muestras era la distribución de Weibull [Wei51], que tiene parámetros $\alpha$, $\beta$, y $\gamma$.

En este contexto, la variable aleatoria bajo estudio, $X$, puede interpretarse como

tiempo de desarrollo. El parámetro de forma $\alpha$ controla el sesgo de la función de distribución de probabilidades, que es asimétrica. Esto se relaciona principalmente con la eficiencia de la organización (EF). El parámetro de escala $\beta$ expande o contrae el gráfico en la dirección de $X$. Este parámetro se relaciona con la eficiencia (EF), la volatilidad de los requerimientos (VR), y la complejidad (CX) medida en CGG. El parámetro de desplazamiento $\gamma$ desplaza el origen de la curva hacia la derecha. Este parámetro se relaciona principalmente con la complejidad medida en CGG.

En base a los resultados de la simulación se concluyó que los parámetros del modelo se pueden derivar de las métricas del proyecto usando el siguiente algoritmo:

```
If (EF >2.0)   then    α = 1.95;
                       γ = 22 * 0.32 * (13 * ln(CGG)− 82);
                       β = γ / (5.71 + (RV− 20) * 0.046);
               else    α = 2.05;
                       γ = 22 * 0.85 * (13 * ln(CGG)− 82);
                       β = γ / (5.47 + (RV− 20) * 0.114);
               end if
```

### 2.6.3. Integración de la Valoración en el Prototipado.

El modelo descripto se diseñó para soportar un proceso iterativo de prototipado y desarrollo de software. En este proceso, una declaración inicial del problema, una versión parcial del prototipo, o un reporte de problemas de un producto de software ya entregado, provocan un análisis del tema, seguido por la formulación de los cambios propuestos a los requerimientos y la especificación de los ajustes propuestos a los requerimientos del software, que al inicio puede estar vacía. En este punto de cada ciclo, el director del proyecto debería ejecutar un paso de valoración de riesgos. El resultado de esta valoración guía el grado de detalle en que se demostrarán las mejoras en los requerimientos, y el conjunto de cuestiones a considerar en el próximo ciclo.

El primer paso de valoración de riesgos basado en las mediciones se puede ejecutar luego de la especificación de la primera versión de la arquitectura del prototipo, en base a los valores obtenidos para la volatilidad de requerimientos, CGG y eficiencia, en las etapas recién concluidas.

En los casos donde la valoración de riesgos se requiere más temprano, antes de que se haya hecho prototipo alguno, las estimaciones de eficiencia y volatilidad de

requerimientos se pueden basar en mediciones de proyectos pasados similares, y la estimación inicial para la complejidad puede basarse en conjeturas subjetivas. Este tipo de estimaciones puede ser menos confiable que las que se basan exclusivamente en las mediciones, pero puede proveer una base razonablemente precisa para decidir si comenzar o no un proceso de prototipado para determinar los requerimientos para un proyecto de desarrollo propuesto. Es decir que partes de este modelo se pueden usar realmente desde el principio del proceso.

Si se aprueba el esfuerzo de prototipado, las mediciones tempranas del proceso podrían usarse para refinar las estimaciones iniciales de los parámetros del modelo, generando así una transición balanceada y sistemática desde la conjetura, codificada como una distribución *a priori*, hacia valoraciones basadas cada vez más en mediciones sistemáticas. Tal enfoque también soporta la incorporación y el refinamiento sistemático de mediciones de ciclos previos del proceso iterativo.

El resultado de la valoración de riesgos puede proveer una guía sobre el grado en el cual el proyecto puede facilitar la exploración de las mejoras en los requerimientos solicitadas por los clientes. También puede ayudar a los clientes o departamento de marketing a decidir cuán necesarias son las posibles mejoras, en el contexto del tiempo resultante y los costos estimados. Un análisis sistemático de costo/beneficio se hace posible sólo con la disponibilidad de estimaciones razonablemente precisas.

El paso de valoración de riesgos puede proveer, por lo tanto, un elemento de balance para estabilizar el proceso de formulación de requerimientos. En ausencia de información acerca del costo de potenciales mejoras, los interesados son propensos a ampliar los requerimientos de forma poco realista – por supuesto, si no se les cobra por las mejoras, siempre desean tener un sistema mejor, no importa cuán bueno sea el que ya tienen. Los pasos propuestos de valoración de riesgos pueden brindar una base realista para incorporar restricciones de tiempo y dinero temprano en el proceso, cuando la situación es fluida y muchas opciones están aún abiertas.

Este refinamiento del proceso provee una comprensión adicional sobre la dinámica del prototipado iterativo: el proceso iterativo debería parar cuando los clientes han determinado qué requerimientos pueden permitirse realizar, y por cuáles de las muchas posibles mejoras estarán dispuestos a pagar. No siempre se da el caso en que el conjunto de críticas surgidas del ciclo final está vacío – esto sólo se da en un mundo ideal con presupuestos adecuados y clientes pacientes.

# Capítulo 3

# Ciclo de Vida Orientado a Objetos

## 3.1. El Ciclo de Vida Orientado a Objetos

### 3.1.1. Conceptos y Principios Orientados a Objetos

Vivimos en un mundo de objetos. Estos objetos existen en la naturaleza, en entidades hechas por el hombre, en los negocios y en los productos que usamos. Ellos pueden ser clasificados, descriptos, organizados, combinados, manipulados y creados. Por esto es natural que se haya propuesto una visión orientada a objetos para la creación de software de computadora, una abstracción que modela el mundo de forma tal que nos ayuda a entenderlo y gobernarlo mejor.

Durante la primera mitad de los años 90, la Ingeniería de Software orientada a objetos se convirtió en el paradigma de elección para muchos desarrolladores de software y un creciente número de profesionales de la ingeniería y sistemas de información. A medida que pase el tiempo las tecnologías de objetos sustituirán a los enfoques clásicos de desarrollo de software, debido a que tienen un número de beneficios inherentes que proporcionan ventajas a los niveles de dirección y técnico.

Por ejemplo, las tecnologías de objetos llevan a reutilizar y la reutilización de componentes de software conduce a un desarrollo de software más rápido y a programas de mejor calidad. El software orientado a objetos es más fácil de mantener debido a que su estructura es inherentemente descompuesta. Esto provoca menores efectos colaterales cuando se deben hacer cambios. Además, los sistemas orientados a objetos son más fáciles de adaptar y escalar.

### El Paradigma Orientado a Objetos

Durante muchos años la expresión "Orientado a Objetos" (OO) se usó para significar un enfoque de desarrollo de software que usaba alguno de los lenguajes de programación orientados a objetos (Ada 95, C++, Eiffel, SmallTalk, Java). Hoy en día el paradigma OO encierra una completa visión de la Ingeniería de Software.

Los beneficios de la tecnología orientada a objetos se fortalecen si se usa antes y durante el proceso de Ingeniería de Software. Un simple uso de programación orientada a objetos no brindará los mejores resultados. Los ingenieros del software y sus directores deben considerar elementos tales como el análisis de requerimientos orientado a objetos, el diseño orientado a objetos, análisis del dominio orientado a objetos, sistemas de gestión de bases de datos orientadas a objetos y la Ingeniería de Software orientada a objetos asistida por computadora [Ber93].

Los sistemas OO tienden a evolucionar con el tiempo. Por esto, un modelo de proceso evolutivo acoplado con un enfoque que fomenta el ensamblaje (reutilización) de componentes es el mejor paradigma para la Ingeniería de Software OO.

El proceso OO se mueve a través de una espiral evolutiva que comienza con la comunicación con el usuario. En esta etapa se logra claridad sobre lo que desea el usuario y la forma en la cual se le va a presentar la solución que está buscando. Aquí se define el dominio del problema y, por medio de la utilización de casos de uso, se identifican las clases básicas del problema. Se individualizan los actores y sus relaciones con los casos de uso. Se describe la información de entrada y salida para cada caso de uso. También, de ser aplicable, se desarrolla una interfaz inicial del sistema. Finalmente, se desarrolla el modelo del mundo, representándolo en un diagrama de estructura estática de clases.

En la siguiente etapa se analizan los requerimientos capturados, refinándolos y estructurándolos. Se logra así una comprensión más precisa de los requerimientos y una descripción de los mismos que es fácil de mantener y que ayuda a estructurar el sistema completo.

En la etapa de Diseño del sistema se define una subdivisión del mismo en aplicaciones (si es lo suficientemente grande) y la forma de comunicación con los sistemas ya existentes con los cuales debe interactuar. Se identifica la arquitectura, definiendo componentes del sistema, las aplicaciones y su ubicación, los mecanismos de comunicación, y se particularizan los casos de uso a la arquitectura planteada.

Durante el Diseño Detallado se adecúa el análisis a las características específicas del ambiente de implementación. Se agregan detalles de implementación al modelo del mundo, se desarrolla el modelo de interfaz, y los modelos de control, persistencia y comunicaciones.

En la etapa de Implementación y Pruebas se desarrolla el código de una manera certificada, se efectúa la codificación y la prueba de módulos unitarios, y por último se comprueba que el sistema funcione correctamente en forma integrada, utilizando para ello casos de prueba diseñados especialmente.

El trabajo técnico asociado con la Ingeniería de Software OO hace hincapié en la reutilización. Por lo tanto, antes de construirse, las clases se buscan en una biblioteca de clases existentes. Incluso si una clase se encuentra en la biblioteca, el desarrollador de software debe aplicar análisis orientado a objetos, diseño orientado a objetos, programación orientada a objetos y pruebas orientadas a objetos para crear o adaptar la clase y los objetos derivados de ella. La nueva clase se pone en la biblioteca de tal manera que pueda ser reutilizada en el futuro.

La visión orientada a objetos demanda un enfoque evolutivo de la Ingeniería de Software. Es excesivamente difícil definir las clases necesarias para un gran sistema o producto en una sola iteración. A medida que el análisis OO y los modelos de diseño evolucionan, se vuelve evidente la necesidad de clases adicionales. Esta es la razón por la que el paradigma arriba descrito trabaja mejor para la OO.

### Conceptos de Orientación a Objetos

**Clases y Objetos.** El bloque de construcción más importante de cualquier sistema orientado a objetos es la *clase*. Una clase es un concepto OO que encapsula las abstracciones de datos y procedimientos que se requieren para describir el contenido y comportamiento de alguna entidad del mundo real o que haya surgido como necesaria en el desarrollo. La Figura 3.1 muestra un ejemplo de clase.

Según Taylor [Tay90], las abstracciones de datos (atributos) que describen la clase están encerradas por una "muralla" de abstracciones procedimentales (llamadas operaciones o servicios) capaces de manipular esos datos de alguna manera. La única forma de alcanzar los atributos (y operar sobre ellos) es ir a través de alguno de los métodos que forman parte de la muralla. Por lo tanto, la clase encapsula datos (dentro de la muralla) y el proceso que manipula los datos (los métodos que componen la

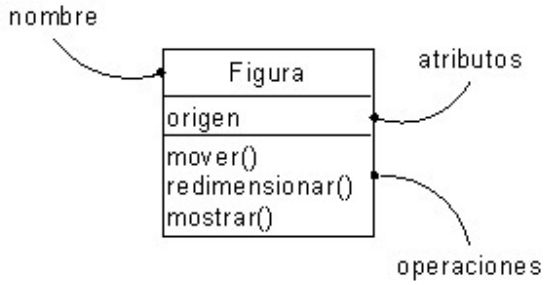


Figura 3.1: Clases

muralla). Esto posibilita el ocultamiento de información y reduce el impacto de efectos colaterales asociados a cambios. Estos métodos tienden a manipular un número limitado de atributos (alta cohesión), y la comunicación ocurre sólo a través de los métodos que encierra la "muralla" (la clase tiende a un bajo acoplamiento con otros elementos del sistema). Todas estas características del diseño conducen al software de alta calidad.

Una clase es una descripción generalizada de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones, y semánticas [BRJ98]. Por definición, todos los objetos que pertenecen a una clase encierran los atributos definidos en la clase y las operaciones disponibles para la manipulación de esos atributos.

Un *objeto* (instancia) es una manifestación concreta de una clase (abstracción) a la cual se le puede aplicar un conjunto de operaciones y que puede tener un estado que almacena los efectos de la operación. Los objetos se usan para modelar cosas concretas o prototípicas que existen en el mundo real o en el sistema de software en ejecución.

Una abstracción denota la esencia ideal de una cosa; una instancia denota una manifestación concreta. En todo lo que se modela se encuentra esta separación entre abstracción e instancia. Para una abstracción dada, se pueden tener innumerables instancias. Para una instancia dada, existe alguna abstracción que especifica las características comunes a todas esas instancias.

**Atributos.** Un *atributo* es una propiedad de una clase, con un nombre, que describe un rango de valores que las instancias de la propiedad pueden tomar. Una clase puede tener cualquier número de atributos, o carecer de ellos (aunque esto no tiene mucho sentido). Un atributo representa alguna propiedad de la cosa que se está mo-

delando que es compartida por todos los objetos de esa clase. Por ejemplo, se podría modelar una clase *Cliente* de forma tal de tener atributos como *nombre*, *dirección*, *teléfono* y *fechaNac*, como se ilustra en la Figura 3.2. Por lo tanto, un atributo es una abstracción del tipo de datos o estado que un objeto de una clase podría abarcar. En un momento dado, un objeto instancia de una clase tendrá valores específicos para cada uno de los atributos definidos en la clase (que en este caso se denominan *campos*) [BRJ98].

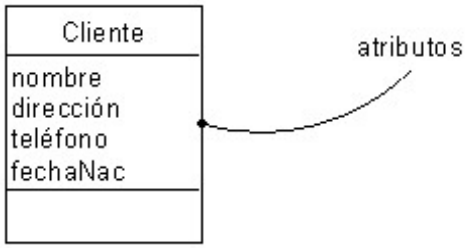


Figura 3.2: Atributos

Un atributo puede verse como una relación binaria entre una clase y cierto dominio. Esta relación binaria implica que un atributo puede tomar un valor definido por un dominio enumerado. En la mayoría de los casos, un dominio es simplemente un conjunto de valores específicos. Por ejemplo, supongamos que una clase *Automóvil* tiene un atributo *color*. El dominio de valores de *color* es {*blanco, negro, plata, gris, azul, rojo, amarillo, verde*}. En situaciones más complejas, el dominio puede ser otra clase. Por ejemplo, la clase *Automóvil* también puede tener un atributo *motor* que abarca los siguientes valores de dominio: {*opción económica, 8 válvulas opción deportiva, 12 válvulas opción deportiva, 16 válvulas opción de lujo*}. Cada una de las opciones indicadas tiene un conjunto de atributos específicos de ella.

**Operaciones.** Una *operación* es la implementación de un servicio que puede ser requerido de cualquier objeto de la clase para afectar el comportamiento. En otras palabras, una operación es una abstracción de algo que se le puede pedir a un objeto que haga, y que es compartido por todos los objetos de esa clase. Una clase puede tener cualquier número de operaciones o carecer de ellas (aunque esto último no tiene mucho sentido). Por ejemplo, todos los objetos de una clase *Rectángulo* podrían ser movidos, redimensionados, o se les podría consultar acerca de sus propiedades. Con frecuencia (pero no siempre) la invocación de una operación sobre un objeto cambia los datos o estado del objeto [BRJ98].

Las operaciones se clasifican en *constructores*, que inicializan el objeto; *comandos*, que cambian el estado interno del objeto y en general no devuelven ningún valor; y *consultas* o *funciones*, que devuelven un valor y no deberían cambiar el estado interno del objeto.

Se puede especificar una operación estableciendo su *signatura*, que comprende el nombre de la operación junto con el nombre, tipo y valor por defecto de todos los parámetros y (en el caso de las funciones) el tipo del valor de retorno, como se ilustra en la Figura 3.3.

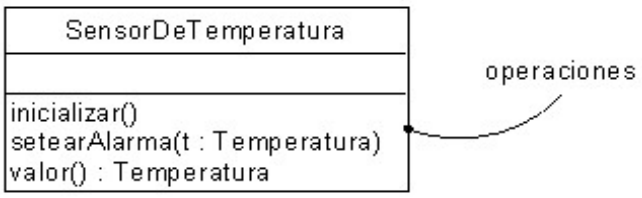


Figura 3.3: Operaciones y sus Signaturas

Cada una de las operaciones encapsuladas por un objeto proporciona una representación de uno de los comportamientos del objeto. Por ejemplo, la operación *consultarColor* para un objeto de la clase *Automóvil* extraerá el color almacenado en el atributo *color*. Cada vez que un objeto recibe un estímulo, éste inicia un cierto comportamiento. Éste puede ser tan simple como determinar el color del coche o tan complejo como la iniciación de una cadena de estímulos que se envían entre una variedad de objetos diferentes.

Se debe hacer una distinción entre operación y *método*. Una operación especifica un servicio que puede ser solicitado a cualquier objeto de la clase para afectar su comportamiento; un método es una implementación de una operación. Una operación puede ser *abstracta*, es decir que puede no tener un método asociado. En este caso, en la clase sólo se especifica su signatura. Si una operación tiene un método asociado, entonces se dice que es no abstracta, o que es *concreta*. Cada operación no abstracta de una clase debe tener un método, que provee un algoritmo ejecutable como cuerpo. En una jerarquía de herencias, podría haber varios métodos para la misma operación, y el polimorfismo y la vinculación dinámica de código (conceptos que se explican en detalle más adelante) eligen cuál método de la jerarquía se despacha en tiempo de ejecución [BRJ98].

**Responsabilidades.** Una *responsabilidad* es un contrato u obligación de una clase. Cuando se crea una clase, se declara que todos los objetos de esa clase tienen el mismo tipo de comportamiento. A un nivel más abstracto, estos atributos y operaciones correspondientes son sólo los medios a través de los cuales se ejecutan las responsabilidades de la clase.

Cuando se modelan clases, es un buen punto de partida especificar las responsabilidades de las cosas en lenguaje natural. Por ejemplo, la Figura 3.4 muestra las responsabilidades de una clase *Despacho*, surgida de un sistema de venta. Una clase puede tener cualquier número de responsabilidades, aunque en la práctica, toda clase bien estructurada tiene como mínimo una responsabilidad y como máximo sólo unas pocas [BRJ98].

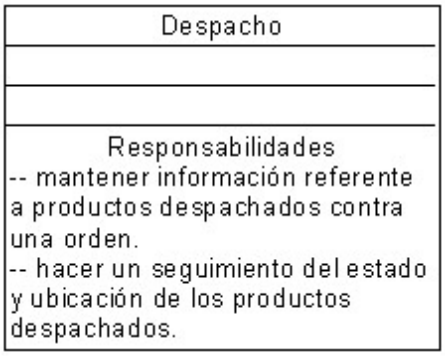


Figura 3.4: Responsabilidades

**Mensajes.** Los *mensajes* son el medio a través del cual interactúan los objetos. Un mensaje estimula la ocurrencia de cierto comportamiento en el objeto receptor. El comportamiento se realiza cuando se ejecuta la operación.

Una operación dentro de un *objeto emisor* (cliente) genera un mensaje de la forma

Mensaje: *destino.operación(parámetros)*

donde *destino* define el *objeto receptor* (servidor) que es estimulado por el mensaje, *operación* se refiere al método que recibe el mensaje, y *parámetros* proporciona información requerida para el éxito de la operación. Por ejemplo, si *sensor1* fuese un objeto de la clase *SensorDeTemperatura* ilustrada en la Figura 3.3, se le podría enviar el mensaje *sensor1.setearAlarma(t)*, donde $t$ es un objeto de clase *Temperatura*, para establecer la temperatura a partir de la cual debería dispararse una alarma.

Cox [Cox86] describe el intercambio entre objetos de la siguiente manera:

> Se solicita a un objeto que ejecute una de sus operaciones enviándole un mensaje que le informa acerca de lo que se debe hacer. El objeto receptor responde al mensaje eligiendo primero la operación que implementa el nombre del mensaje, ejecutando luego dicha operación y devolviendo, por último, el control al objeto que origina la llamada.

El paso de mensajes mantiene comunicado un sistema orientado a objetos. Los mensajes proporcionan una visión interna del comportamiento de objetos individuales, y del sistema OO como un todo.

**Encapsulamiento, Herencia y Polimorfismo.** Estas tres características hacen únicos a los sistemas orientados a objetos. Como ya hemos observado, las clases OO y los objetos derivados de ellas *encapsulan* los datos y las operaciones que trabajan sobre éstos en un único paquete. Esto proporciona un número importante de beneficios:

- Los detalles de implementación interna de datos y procedimientos están ocultos al mundo exterior (ocultamiento de la información). Esto reduce la propagación de efectos colaterales cuando ocurren cambios.
- Las estructuras de datos y las operaciones que las manipulan están mezcladas en una única entidad: la clase. Esto facilita la reutilización de componentes.
- Las interfaces entre objetos encapsulados están simplificadas. Un objeto que envía un mensaje no tiene que preocuparse por los detalles de las estructuras de datos internas en el objeto receptor. Por tanto se simplifica la interacción, y el acoplamiento del sistema tiende a reducirse.

La *herencia* (también llamada generalización) es una relación entre una cosa general (llamada superclase o clase padre) y un tipo más específico de esa cosa (llamada subclase o clase hija). La generalización es también llamada relación "es un tipo de": una cosa (como la clase *Cuadrado*) "es un tipo de" una cosa más general (por ejemplo, la clase *Rectángulo*). Esta relación se ilustra en la Figura 3.5, donde se observa que *Rectángulo*, junto con otras clases (*Círculo* y *Polígono*), hereda a su vez de una clase más general: *Figura*. La generalización significa que los objetos de la subclase pueden ser usados en cualquier lugar donde aparezca un objeto instancia de la superclase, pero no a la inversa. En otras palabras, el hijo es un sustituto para el padre [BRJ98].

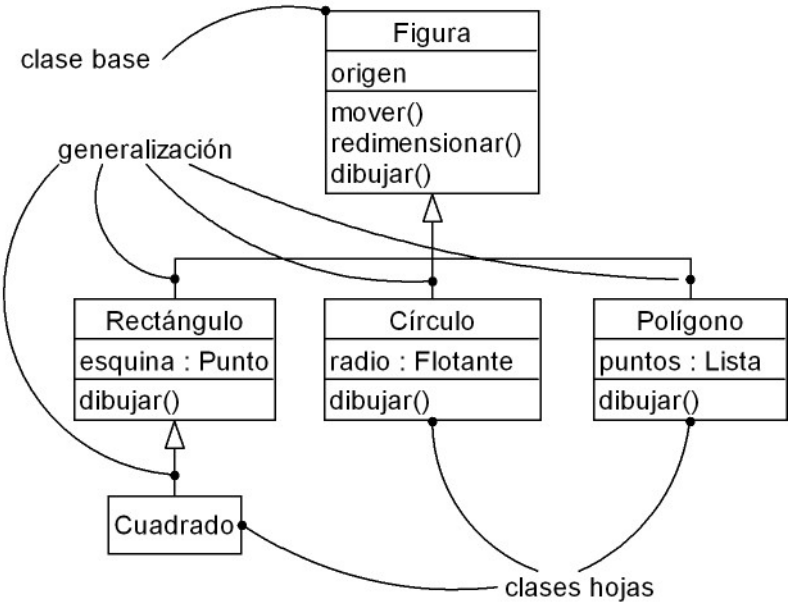


Figura 3.5: Herencia o Generalización

La clase hija hereda las propiedades de la clase padre, especialmente sus atributos y operaciones. Esto significa que todas las estructuras de datos y algoritmos originalmente diseñados e implementados para la superclase están inmediatamente disponibles para la subclase (no es necesario más trabajo extra). La reutilización se realiza directamente. Cualquier cambio en los datos u operaciones contenidas dentro de una superclase se hereda inmediatamente por todas las subclases que se derivan de la superclase. Debido a esto, la jerarquía de clases se convierte en un mecanismo a través del cual los cambios (a altos niveles) pueden propagarse inmediatamente a través de todo el sistema.

Es importante destacar que en cada nivel de la jerarquía de clases, pueden añadirse nuevos atributos y operaciones a aquellos que han sido heredados de niveles superiores de la jerarquía. Además, y debido a que la reestructuración de la jerarquía puede ser difícil, se usa a veces la *anulación*. En esencia, la anulación ocurre cuando los atributos y operaciones se heredan de manera normal, pero después son modificados según las necesidades específicas de la nueva clase.

En algunos casos es tentador heredar algunos atributos y operaciones de una clase y otros de otra clase. Esta acción se llama *herencia múltiple*, y es controvertida. En general, la herencia múltiple complica la jerarquía de clases y crea problemas potenciales en el control de la configuración. Los cambios en la definición de una clase

que reside en la parte superior de la jerarquía pueden tener impactos no deseados originalmente en las clases definidas en zonas inferiores de la arquitectura.

En POO se llama *polimorfismo* a la propiedad que tienen los métodos de mantener una respuesta unificada, con la misma semántica, aunque con distinta implementación, a través de la jerarquía de clases. O dicho de otra manera, la llamada a una misma operación puede invocar a métodos distintos. A veces se lo denomina *abstracción de mensajes*.

El polimorfismo logra retardar la decisión sobre el tipo (o clase) del objeto hasta el momento en que vaya a ser utilizado el método. En este sentido, el polimorfismo está asociado a lo que se denomina *vinculación tardía* o *vinculación en tiempo de ejecución*. La idea es que los objetos de distintas clases puedan ser tratados de la misma manera, y se le apliquen los mismos métodos, aunque las implementaciones particulares sean diferentes. Esto permite una mayor separación entre interfaz e implementación y facilita la extensión del código, ya que una vez que se escribe un método para una clase, la creación de clases derivadas no implica redefinirlo si su semántica es similar.

Las operaciones tienen propiedades similares. Toda operación es polimórfica, lo que significa que, en una jerarquía de clases, es posible especificar operaciones con la misma signatura en diferentes puntos de la jerarquía. Las operaciones en las clases hijas sustituyen el comportamiento de las pertenecientes a las clases superiores. Por ejemplo, en la Figura 3.5 la operación *dibujar()* es polimórfica, ya que aparece con la misma signatura tanto en la clase *Figura* como en las clases hijas *Rectángulo*, *Círculo* y *Polígono*.

La *redefinición* es una funcionalidad de la POO que se usa cuando el comportamiento de la clase ancestro no es exactamente igual para la descendiente. Por ejemplo, en el caso de las figuras geométricas, el método *dibujar* puede no ser igual para todas las clases, aunque todas dibujen una figura. En ese caso, habría un método *dibujar* para *Figura*, otro para *Rectángulo*, otro para *Círculo*, etc. Simplemente, esto se resuelve escribiendo diferentes métodos *dibujar*, y se dice que el método *dibujar* de una clase descendiente redefine al método *dibujar* de su ancestro. Esta redefinición se debe a que ahora, cada vez que se invoque al método *dibujar* para una instancia de *Círculo* nos estaremos refiriendo, no ya al que se hubiera heredado de *Figura*, sino al redefinido en *Círculo*.

### Gestión de Proyectos de Software Orientado a Objetos

La moderna gestión de proyectos de software puede subdividirse en las siguientes actividades [Pre97]:

1. Establecimiento de un marco de proceso común para el proyecto.
2. Uso del marco y de métricas históricas para desarrollar estimaciones de esfuerzo y tiempo.
3. Especificación de productos de trabajo y avances que permitirán la medición del progreso.
4. Definición de puntos de comprobación para asegurar la calidad y el control.
5. Gestión de los cambios que ocurren inevitablemente al progresar el proyecto.
6. Seguimiento, monitoreo y control del progreso.

El director técnico que se enfrente con un proyecto de software orientado a objetos aplica estas seis actividades. Pero debido a la naturaleza única del software orientado a objetos, cada una de estas actividades de gestión tiene un matiz levemente diferente y debe ser enfocado usando un modelo propio. Los principios de gestión fundamentales serán los mismos, pero la técnica debe ser adaptada de tal manera que un proyecto OO sea dirigido correctamente.

#### *El Marco de Proceso Común para OO*

Un *marco de proceso común (MPC)* define un enfoque organizativo para el desarrollo y mantenimiento de software. El MPC identifica el paradigma de Ingeniería de Software aplicado para construir y mantener el software, así como las tareas, hitos y entregas requeridos. Establece el grado de rigor con el cual se enfocarán los diferentes tipos de proyectos.

El MPC adaptable, de tal manera que siempre cumpla con las necesidades individuales del equipo del proyecto. Ésta es su característica más importante. Un MPC efectivo para proyectos OO no es un modelo lineal secuencial. Por su naturaleza, la Ingeniería de Software orientada a objetos debe aplicar un paradigma que contemple el desarrollo iterativo. Esto es, el software OO evoluciona a través de un número

de ciclos. El marco de proceso común usado para dirigir un proyecto OO debe ser evolutivo por naturaleza.

Ed Berard [Ber93] y Grady Booch [Boo91], entre otros, sugieren el uso de un "modelo recursivo/paralelo" para el desarrollo de software orientado a objetos. En esencia, el modelo recursivo/paralelo prevee los siguientes pasos:

- Realizar los análisis suficientes para aislar las clases del problema y las conexiones más importantes.
- Realizar un pequeño diseño para determinar si las clases y conexiones pueden ser implementadas de manera práctica.
- Extraer objetos reutilizables de una biblioteca para construir un prototipo preliminar.
- Conducir algunas pruebas para descubrir errores en el prototipo.
- Obtener realimentación del cliente sobre el prototipo.
- Modificar el modelo de análisis basándose en lo que se ha aprendido del prototipo, de la realización del diseño y de la realimentación obtenida del cliente.
- Refinar el diseño para acomodar sus cambios.
- Construir objetos especiales (no disponibles en la biblioteca).
- Ensamblar un nuevo prototipo usando objetos de la biblioteca y los objetos que se crearon nuevos.
- Realizar pruebas para descubrir errores en el prototipo.
- Obtener realimentación del cliente sobre el prototipo.

Este enfoque continúa hasta que el prototipo evoluciona hacia una aplicación productiva.

Berard [Ber93] describe el modelo de la siguiente manera:

- Descomponer sistemáticamente el problema en componentes altamente independientes.

- Aplicar de nuevo, si fuese necesario, el proceso de descomposición a cada una de las componentes independientes para, a su vez, descomponerlas (la parte recursiva).
- Ejecutar este proceso de reaplicar la descomposición simultáneamente sobre cada una de las componentes (la parte paralela).
- Continuar este proceso hasta cumplir los criterios de completitud.

Es importante observar que el proceso de descomposición mostrado anteriormente se cancela si el analista/diseñador reconoce que la componente o subcomponente requerida está disponible en una biblioteca de reutilización.

Las tareas y la planificación del proyecto están unidas a cada una de las "componentes altamente independientes", y el progreso se mide individualmente para cada una de estas componentes. Cada iteración del proceso recursivo/paralelo requiere planificación, ingeniería (análisis, diseño, extracción de clases, prototipado y pruebas) y actividades de evaluación.

Consideremos, por ejemplo, que se nos ha encomendado la tarea de crear un "sistema de mensajería electrónica" (SME). ¿Cuáles serían algunos escenarios para el desarrollo orientado a objetos de un sistema como este?

Un escenario podría ser la afirmación de que:

- El SME se compone de:
  - una Oficina Postal (espacio en el servidor donde se guardan los mensajes a la espera de que el destinatario los baje),
  - un portador de mensajes,
  - y otros objetos del SME...
- La Oficina Postal, a su vez, se compone de:
  - un empleado postal,
  - una colección de buzones (casillas de correo),
  - y otros objetos de la oficina postal...
- Uno de los componentes de una colección de buzones es un buzón particular.

- Uno de los componentes de un buzón particular sería un mensaje individual.
- Los componentes de un mensaje podrían ser:
  - el nombre del remitente,
  - la dirección del remitente,
  - el nombre del destinatario,
  - la dirección del destinatario,
  - el matasellos (es decir, la fecha y hora en que se envió el mensaje),
  - y el texto en sí del mensaje.

Dado el SME anteriormente descripto (parcialmente):

- El primer paso del Análisis de Requerimientos Orientado a Objetos debería:
  - identificar la oficina postal, el portador de mensajes, y los "otros objetos" como componentes del SME,
  - describir el entorno en el cual se usará el SME, y
  - describir las interfaces conceptuales del SME, de la oficina postal, del portador de mensajes, y de los "otros objetos".
- El primer paso del Diseño Orientado a Objetos debería:
  - detallar las interfaces precisas en el lenguaje de programación para el SME, la oficina postal, el portador de mensajes, y los "otros objetos".
  - crear el algoritmo en lenguaje de programación para la estructura interna del SME en su nivel más alto, es decir cómo el SME manipulará la oficina postal, el portador de mensajes, y los "otros objetos".
- Asumiendo que el portador de mensajes es medianamente simple, el primer paso de la Programación Orientada a Objetos crearía la totalidad del código fuente para el portador (y para cualquier otro objeto simple de ese nivel).
- Asumiendo un cierto nivel de complejidad, uno de los segundos pasos del Análisis de Requerimientos identificaría la colección de buzones, el empleado postal, y los "otros objetos de la oficina postal" como componentes de la oficina postal.

- El paso de Diseño para la oficina postal debería:
  - detallar las interfaces precisas en lenguaje de programación para la colección de buzones, el empleado postal, y los "otros objetos de la oficina postal".
  - crear el algoritmo en lenguaje de programación para la estructura interna de la oficina interna en su nivel más alto, es decir cómo la oficina postal manipulará la colección de buzones, el empleado postal, y los "otros objetos de la oficina postal".
- Asumiendo que el empleado postal es un objeto medianamente simple, la Programación crearía la totalidad del código fuente para el empleado postal.

Volviendo al nivel más alto de abstracción (es decir, el SME):

- La oficina postal, el portador de mensajes, y los "otros objetos" serían los "objetos de análisis" para el SME (objetos identificados durante el análisis que no se convierten a código fuente).
- Con respecto al SME, la colección de buzones, el empleado postal, y los "otros objetos de la oficina postal" son "objetos de diseño" (objetos identificados durante el diseño que no fueron tratados en el análisis, y que casi con seguridad se convertirán en código fuente).
- La oficina postal, el portador de mensajes y los "otros objetos" son "objetos de interfaz" con respecto al SME y al resto del sistema (objetos que, tomados en conjunto, constituyen la interfaz de usuario para el sistema de objetos).
- Los objetos mensaje son los "objetos que se desplazan" con respecto al SME de nivel superior y a la oficina postal (objetos que fueron mencionados en el análisis *y* son necesarios para el adecuado comportamiento del sistema de objetos. Estos objetos son entradas al sistema de objetos, salidas del sistema de objetos, o ambas cosas).

***Seguimiento del Progreso en un Proyecto Orientado a Objetos***

Aunque el modelo de proceso recursivo/paralelo es el mejor marco de trabajo para un proyecto OO, el paralelismo de tareas dificulta el seguimiento del proyecto. El jefe

del proyecto puede tener dificultades para establecer hitos significativos en un proyecto OO debido a que cierto número de cosas están ocurriendo a la vez. En general, los siguientes hitos pueden considerarse "completos" (terminados) al cumplirse los criterios mostrados:

- ***Hito técnico: Análisis OO terminado***
  - Todas las clases, y la jerarquía de clases, están definidas y revisadas.
  - Se han definido y revisado los atributos de clases y las operaciones asociadas a una clase.
  - Se han establecido y revisado las relaciones entre clases.
  - Se ha creado y revisado un modelo de comportamiento.
  - Se han marcado clases reutilizables.
- ***Hito técnico: Diseño OO terminado***
  - Se ha definido y revisado el conjunto de subsistemas.
  - Las clases se han asignado a subsistemas y han sido revisadas.
  - Se ha establecido y revisado la asignación de tareas.
  - Se han identificado responsabilidades y colaboraciones.
  - Se han diseñado y revisado los atributos y operaciones.
  - El modelo de mensajería (pasaje de mensajes) ha sido creado y revisado.
- ***Hito técnico: Programación OO terminada***
  - Cada nueva clase ha sido implementada en código a partir del modelo de diseño.
  - Se han integrado las clases extraídas de bibliotecas de reutilización.
  - Se ha construido un prototipo o incremento.
- ***Hito técnico: Prueba OO***
  - La corrección y completitud del análisis OO y del modelo de diseño han sido revisados.

- Se han desarrollado y revisado una red de clases–responsabilidades–colaboraciones.
- Se han diseñado casos de prueba y ejecutado pruebas a nivel de clases para cada clase.
- Se han diseñado casos de prueba y completado pruebas de agrupamiento, y las clases se han integrado.
- Las pruebas a nivel de sistema se han terminado.

Recordando el modelo de proceso recursivo/paralelo examinado anteriormente, es importante destacar que cada uno de estos hitos puede ser visitado nuevamente al entregar diferentes incrementos al usuario.

### 3.1.2. Análisis Orientado a Objetos

El objetivo del Análisis Orientado a Objetos (AOO) es desarrollar una serie de modelos que describan el software de computadora necesario para satisfacer un conjunto de requerimientos definidos por el cliente. Durante el AOO se definen todas las clases (y las relaciones y comportamientos asociados con ellas) que son relevantes al problema que se va a resolver. Para cumplirlo se deben ejecutar las siguientes tareas:

1. Obtener del cliente los requerimientos básicos del usuario.
2. Identificar las clases (es decir, definir atributos y métodos).
3. Especificar una jerarquía de clases.
4. Representar las relaciones objeto a objeto (conexiones de objetos).
5. Modelar el comportamiento del objeto.
6. Repetir iterativamente las tareas de la 1 a la 5 hasta completar el modelo

Para construir un modelo de análisis se aplican cinco principios básicos [Pre97]: (1) se modela el dominio de la información; (2) se describe la función del módulo; (3) se representa el comportamiento del modelo; (4) los modelos se dividen para mostrar más detalles; y (5) los modelos iniciales representan la esencia del problema mientras que los últimos aportan detalles de la implementación.

### Enfoques convencionales y enfoques OO

El enfoque del AOO representa un cambio radical sobre las metodologías orientadas a procesos como el análisis estructurado, pero sólo un cambio incremental respecto a las metodologías orientadas a datos como la ingeniería de la información. Las metodologías orientadas a procesos desvían la atención de las prioridades inherentes a los objetos durante el proceso de modelado, y conducen a un modelo del dominio del problema que es ortogonal con los tres principios esenciales de la orientación a objetos: encapsulamiento, clasificación de objetos y herencia [FK92].

El análisis estructurado toma una visión diferente de los requerimientos del modelo entrada–proceso–salida. Los datos se consideran separadamente de los procesos que los transforman. El comportamiento del sistema, aunque importante, tiende a jugar un papel secundario en el análisis estructurado. El enfoque del análisis estructurado hace un fuerte uso de la descomposición funcional.

La popularidad de las tecnologías de objetos ha generado docenas de métodos de AOO. Cada uno de ellos introduce un proceso para el análisis de un producto o sistema, un conjunto de modelos que evoluciona junto con el proceso, y una notación que posibilita al ingeniero del software crear cada modelo de una manera consistente. Pero si bien la terminología y las etapas del proceso varían de un método a otro, los procesos generales de AOO son en realidad muy similares.

Para realizar un análisis orientado a objetos, un ingeniero de software debería ejecutar las siguientes etapas genéricas:

- Obtener los requerimientos del cliente para el sistema OO.
  - Identificar escenarios o casos de uso.
  - Construir un modelo de requerimientos.
- Seleccionar clases y objetos usando los requerimientos básicos como guía.
- Identificar atributos y operaciones para cada objeto del sistema.
- Definir estructuras y jerarquías que organicen las clases.
- Construir un modelo objeto–relación.
- Construir un modelo objeto–comportamiento.
- Revisar el modelo de análisis OO en relación a los casos de uso/escenarios.

### Componentes Genéricos del Modelo de Análisis OO

Aunque la terminología, notación y actividades difieren respecto de los usados en métodos convencionales, el AOO (en su núcleo) resuelve los mismos objetivos subyacentes. Dice Rumbaugh [R+91]: "El análisis se ocupa de proyectar un modelo preciso, conciso, comprensible y correcto del mundo real. El propósito del AOO es modelar el mundo real de forma tal que sea comprensible. Para esto se deben examinar los requerimientos, analizar las implicaciones que se deriven de ellos y reafirmar de manera rigurosa. Se deben abstraer primero características del mundo real y dejar los pequeños detalles para más tarde". Para desarrollar un "modelo preciso, conciso, comprensible y correcto del mundo real", un ingeniero de software debe seleccionar una de un cierto número de notaciones y procesos del AOO.

Monarchi y Puhr [MP92] definen un conjunto de componentes de representación genéricos que aparecen en todos los modelos de AOO. Los *componentes estáticos* son estructurales por naturaleza, e indican características que se mantienen durante toda la vida operacional de una aplicación. Los *componentes dinámicos* se centran en el control, y son sensibles al tiempo y al tratamiento de eventos. Ellos definen cómo interactúa un objeto con otros a lo largo del tiempo. Se pueden identificar los siguientes componentes:

**Vista estática de clases semánticas.** Se imponen los requerimientos y se extraen (y representan) clases como parte del modelo de análisis. Estas clases persisten a través de todo el período de vida de la aplicación y se derivan en base a la semántica de los requerimientos del cliente.

**Vista estática de los atributos.** Los atributos asociados con la clase aportan una descripción de la clase, así como una indicación inicial de las operaciones relevantes a esta clase.

**Vista estática de las relaciones.** El modelo de análisis debe representar las relaciones para que puedan identificarse las operaciones (que afectan estas conexiones) y para que pueda desarrollarse un buen diseño de intercambio de mensajes.

**Vista estática de los comportamientos.** Las relaciones indicadas anteriormente definen un conjunto de comportamientos que se adaptan al escenario utilizado (casos de uso) del sistema.

**Vista dinámica de la comunicación.** Los objetos deben comunicarse unos con otros y hacerlo basándose en una serie de mensajes que provoquen transiciones de un estado a otro del sistema.

**Vista dinámica del control y manejo del tiempo.** Debe describirse la naturaleza y tiempo de duración de los eventos que provocan transiciones de estados.

Los componentes estáticos y dinámicos se identifican para el objeto internamente y para las representaciones interobjetos. Una vista dinámica del objeto internamente puede caracterizarse como la *historia de vida del objeto*, esto es, los estados que alcanza el objeto a lo largo del tiempo, al realizarse una serie de operaciones sobre sus atributos.

### El Proceso de AOO

El proceso de AOO comienza con una comprensión de la manera en la que se usará el sistema: por las personas, por las máquinas, o por otros programas. Una vez que se ha definido el escenario, comienza el modelado del software.

#### *Casos de Uso (Use Cases)*

La captura de requerimientos es siempre el primer paso en cualquier actividad de análisis del software. Basado en estos requerimientos, el ingeniero de software (analista) puede crear un conjunto de escenarios de manera tal que cada uno identifique una parte del uso que se le dará al sistema a construir. Los escenarios, a menudo llamados *casos de uso* [Jac92], aportan una descripción acerca de cómo será usado el sistema.

Para crear un caso de uso, el analista debe primero identificar los diferentes *actores* que usan el sistema (personas o dispositivos). Un actor es cualquier cosa que se comunique con el sistema o producto y que sea externo a él. Es importante observar que un actor y un usuario *no* son la misma cosa. Un usuario típico puede desempeñar un cierto número de *roles* cuando usa el sistema, mientras que el actor representa una clase de entidades externas (a menudo, pero no siempre, las personas) que sólo desempeñan un único papel.

Durante la primera iteración no se identifican todos los actores. En general, es posible identificar *actores primarios* durante esa primera iteración, y *actores secundarios* al aprender más sobre el sistema. Los actores primarios interactúan para lograr el funcionamiento requerido del sistema y obtener de él el beneficio propuesto. Ellos

trabajan directa y frecuentemente con el software. Los actores secundarios existen para dar soporte al sistema de manera tal que los primarios puedan realizar su trabajo. Por ejemplo, en un caso de uso que involucre la compra/venta de un bien o servicio, los actores primarios son el comprador y el vendedor del bien o servicio, mientras que si el pago se realiza mediante un depósito bancario, el empleado del banco constituye un actor secundario para el caso de uso.

Una vez identificados los actores primarios, pueden desarrollarse los casos de uso. Un caso de uso describe la forma en la cual un actor interactúa con el sistema. En general, un caso de uso es simplemente una narración escrita que detalla el papel de un actor al interactuar con el sistema.

Cada caso de uso deberá revisarse cuidadosamente. Si algún elemento de la interacción es ambiguo, es probable que una revisión del caso de uso revelará un problema. Cada caso de uso aporta un escenario no ambiguo de interacción entre un actor y el software. Pueden también usarse para especificar requerimientos de tiempo u otras restricciones para el escenario. Los casos de uso describen escenarios que pueden percibirse de manera diferente por diferentes actores.

***Modelado de clases–responsabilidades–colaboraciones***

Una vez que se han desarrollado los escenarios de uso básicos para el sistema, es tiempo de identificar las clases candidatas, e indicar sus responsabilidades y colaboraciones. El modelado de clases–responsabilidades–colaboraciones (CRC) [WB+90] aporta un medio sencillo de identificar y organizar las clases que resulten relevantes al sistema o a los requerimientos del producto.

Un modelo CRC es en realidad un conjunto de tarjetas índice estándar que representan clases [Amb95]. Cada tarjeta se divide en tres secciones: en el encabezado se escribe el nombre de la clase, y en el cuerpo se listan las responsabilidades de la clase a la izquierda, y los colaboradores a la derecha, como lo muestra la Figura 3.6. Estas tarjetas pueden ser reales o virtuales.

Las *responsabilidades* son los atributos y operaciones relevantes para la clase. Una responsabilidad es "cualquier cosa que conoce o hace la clase" [Amb95]. Los colaboradores son aquellas clases necesarias para proveer a una clase con la información necesaria para completar una responsabilidad. En general, una colaboración implica una solicitud de información o de alguna acción.

**Clases**

Los objetos se manifiestan en una variedad de formas: entidades externas, cosas, ocurrencias o eventos, roles, unidades organizacionales, lugares, o estructuras. Una técnica para identificarlos en el contexto de un problema del software es realizar un análisis gramatical con la narrativa de procesamiento para el sistema. Todos los sustantivos se transforman en objetos potenciales. Sin embargo, no todo objeto potencial podrá incluirse en el modelo CRC. Se han definido seis características de selección: información retenida, servicios necesarios, múltiples atributos, atributos comunes, operaciones comunes y requerimientos esenciales. Un objeto potencial debe satisfacer estas seis características para poder ser considerado como posible miembro del modelo CRC.

| Nombre de la clase: | |
|---|---|
| Tipo de Clase: (dispositivo, propiedad, rol, evento, ...) | |
| Características de la clase: (tangible, atómica, concurrente, ...) | |
| Responsabilidades: | Colaboradores: |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

Figura 3.6: Modelo CRC de tarjeta índice

Adicionalmente, los objetos y clases pueden clarificarse por un conjunto de propiedades:

**Tangibilidad.** ¿Representa la clase algo tangible o palpable, o representa información más abstracta?

**Inclusividad.** ¿Es la clase *atómica* o *agregada*?

**Secuenciabilidad.** ¿Es la clase *concurrente* o *secuencial*?

**Persistencia.** ¿Es la clase *transitoria*, *temporal* o *permanente*?

**Integridad.** ¿Es la clase corrompible o es segura?

**Responsabilidades**

Se denominan responsabilidades a los atributos y operaciones de una clase. Los atributos representan características estables de una clase. A menudo pueden extraerse del planteamiento de alcance, o discernirse a partir de la comprensión de la naturaleza de la clase. Las operaciones pueden extraerse desarrollando una análisis gramatical sobre la narrativa de procesamiento del sistema. Los verbos se transforman en candidatos a operaciones.

Wirfs-Brock y sus colegas [WB+90] sugieren cinco pautas para especificar responsabilidades para las clases:

1. *La inteligencia del sistema debe distribuirse de manera igualitaria.* Toda aplicación encierra un cierto grado de inteligencia, por ejemplo, lo que sabe el sistema, las acciones que puede ejecutar, y el impacto que tiene sobre otros sistemas y sus usuarios. Dados sus roles dentro de un sistema, algunos objetos pueden verse como más inteligentes que otros. Un objeto incorpora más o menos inteligencia de acuerdo a cuánto sabe o puede hacer y a cuántos otros objetos afecta. El objetivo no es distribuir uniformemente la inteligencia, sino darles a los objetos las responsabilidades que puedan manejar.

2. *Cada responsabilidad debe establecerse lo más general posible.* No hay que ser demasiado específico en la enunciación de las responsabilidades. Si esta enunciación se redacta en forma general, puede abarcar muchas demandas específicas. Además, no hay espacio suficiente en una tarjeta CRC para registrar demasiados detalles, así que hay que usarlas con sensatez.

3. *La información y el comportamiento asociado a ella, deben encontrarse dentro de la misma clase.* Esto implementa el principio llamado encapsulamiento. Si un objeto es responsable de mantener cierta información, es lógico asignarle responsabilidades para ejecutar las operaciones sobre esa información.

4. *La información sobre un elemento debe estar localizada dentro de una clase, no distribuida a través de varias clases.* En general, es más fácil cumplir con la responsabilidad de mantener información específica si esa información no está compartida. El compartir implica una duplicación que puede dar lugar a inconsistencias.

5. *Compartir responsabilidades entre clases relacionadas cuando sea apropiado.* Esto se refiere a casos en que varios objetos relacionados deben exhibir todos el mismo comportamiento al mismo tiempo. Por ejemplo, se les podría solicitar a todas las clases de una estructura todo-parte que provean un servicio (que asuman parte de una responsabilidad) para responder a un evento externo.

**Colaboradores**

Las colaboraciones representan solicitudes de un cliente a un servidor en el cumplimiento de una responsabilidad del cliente. Una colaboración es la realización de un contrato entre el cliente y el servidor. Decimos que un objeto colabora con otro si para ejecutar una responsabilidad necesita enviar cualquier mensaje al otro objeto. Una colaboración simple fluye en una dirección, representando una solicitud del cliente al servidor. Desde el punto de vista del cliente, cada una de sus colaboraciones está asociada con una responsabilidad particular implementada por el servidor [WB$^+$90].

Las colaboraciones identifican relaciones entre clases. Cuando todo un conjunto de clases colabora para satisfacer algún requisito, es posible organizarlas en un subsistema (un elemento del diseño). Las colaboraciones se identifican determinando si una clase puede satisfacer cada responsabilidad. Si no puede, entonces necesita interactuar con otra clase. Por consiguiente, necesita una colaboración.

Para ayudar en la identificación de colaboradores, el analista puede examinar tres relaciones genéricas diferentes entre clases: (1) la relación *es-parte-de*, (2) la relación *tiene-conocimiento-sobre*, y (3) la relación *depende-de*. A través de la creación de un diagrama de relación entre clases el analista desarrolla las conexiones necesarias para identificar estas relaciones.

Todas las clases que forman parte de una clase agregada, están conectadas a ésta a través de una relación *es-parte-de*. Cuando una clase debe obtener información sobre otra, se establece la relación *tiene-conocimiento-sobre*. La relación *depende-de* implica que dos clases poseen una dependencia no realizable a través de *tiene-conocimiento-sobre* o *es-parte-de*.

El modelo CRC es una primera representación del modelo de análisis para un sistema OO. Puede ser "comprobado" realizando una revisión dirigida por los casos de uso derivados del sistema.

### Definición de estructuras y jerarquías

Una vez que se han identificado las clases y objetos usando el modelo CRC, debe derivarse una estructura de *generalización–especialización* para las clases identificadas [CY91]. En otros casos, un objeto representado según el modelo inicial puede estar compuesto realmente de un número de partes las cuales pueden definirse a su vez como objetos. Estos objetos agregados pueden representarse como una estructura *todo–partes*.

Las representaciones estructurales proveen al analista de los medios para particionar el modelo CRC y para representar esta partición gráficamente. La expansión de cada clase aporta los detalles necesarios para revisión y para el subsiguiente diseño.

#### *Definición de temas y subsistemas*

Un modelo de análisis para una aplicación compleja puede tener cientos de clases y docenas de estructuras. Por esta razón, es necesario definir una representación concisa que sea un resumen de los modelos CRC y estructural descriptos anteriormente.

Los subconjuntos de clases que colaboran entre sí para llevar a cabo un conjunto de responsabilidades cohesionadas son conocidos normalmente como *temas* o *subsistemas*. Ambos, temas y subsistemas, son abstracciones que aportan una referencia o puntero a los detalles en el modelo de análisis. Un subsistema implementa uno o más *contratos* con sus colaboradores externos. Un contrato es una lista específica de solicitudes que los colaboradores pueden hacer a un subsistema.

Los temas son idénticos a los subsistemas en intención y contenido, pero se representan gráficamente. Las referencias de temas se crean generalmente para cualquier estructura que posea cinco o seis objetos. Al nivel más abstracto, el modelo de AOO contendrá solamente referencias de temas. Cada una de las referencias se expandirá a una estructura.

### El modelo Objeto–Relación

Existe una *relación* cuando dos clases cualesquiera están conectadas. Debido a esto los colaboradores siempre están relacionados de alguna manera. El tipo de relación más común es la binaria. Una relación binaria posee una dirección específica que se define a partir de qué clase desempeña el papel de cliente y cuál actúa como servidor.

Las relaciones pueden derivarse a partir del examen de los verbos o frases verbales

en el establecimiento del alcance o casos de uso para el sistema. Usando un análisis gramatical, el analista aísla expresiones que indican localizaciones físicas o emplazamientos (*cerca de*, *parte de*, *contenido en*), comunicaciones (*transmite a*, *obtenido de*), propiedad (*incorporado por*, *se compone de*) y cumplimiento de una condición (*dirige*, *coordina*, *controla*). Estos verbos aportan una indicación de la relación [R$^+$91].

El modelo objeto–relación (como el modelo entidad–relación) puede derivarse en tres pasos o etapas:

1. Usando las tarjetas índice CRC, puede dibujarse una red de objetos colaboradores. Primero se dibujan los objetos conectados por líneas sin etiquetas.

2. Revisando el modelo CRC de tarjetas índice, se evalúan responsabilidades y colaboradores, y cada línea de conexión sin etiquetar recibe un nombre. Con una punta de flecha se indica la *dirección* de la relación.

3. Una vez que se han establecido y nombrado las relaciones, se evalúa cada extremo para determinar la cardinalidad.

Los pasos anteriormente mostrados continúan hasta que se produzca un modelo objeto–relación completo.

En el desarrollo de un modelo objeto–relación, el analista añade aún alguna otra dimensión al modelo de análisis general. No solamente se identifican las relaciones entre objetos, sino que se definen todas las vías importantes de mensajes.

### El modelo Objeto–Comportamiento

Ahora es el momento para hacer una transición al comportamiento dinámico del sistema o producto OO. Debemos representar el comportamiento del sistema como una función de eventos específicos y tiempo.

El modelo *objeto–comportamiento* indica cómo responderá un sistema OO a eventos o estímulos externos. Para crear el modelo, el analista debe ejecutar los siguientes pasos:

1. Evaluar todos los casos de uso para comprender totalmente la secuencia de interacción dentro del sistema.

2. Identificar eventos que dirigen la secuencia de interacción y comprender cómo se relacionan estos eventos con objetos específicos.

3. Crear una traza de eventos para cada caso de uso.

4. Construir un diagrama de transición de estados para el sistema.

5. Revisar la exactitud y consistencia del modelo objeto–comportamiento.

***Identificación de eventos con casos de uso***

En general, un *evento* ocurre cada vez que un sistema OO y un actor intercambian información. Un evento es booleano. Esto es, un evento *no* es la información que se intercambia; es el hecho de que la información ha sido intercambiada. Un caso de uso se examina por puntos de intercambio de información.

Deberá identificarse un actor para cada evento; debe anotarse la información que se intercambia, y deberán indicarse otras condiciones o restricciones. Una vez que todos los eventos han sido identificados, se asocian los objetos incluidos.

**Representaciones de estados**

En el contexto de sistemas OO, deben considerarse dos caracterizaciones de estados:

- El estado de cada objeto cuando el sistema ejecuta su función.
- El estado del sistema observado desde el exterior cuando éste ejecuta su función.

El estado de un objeto adquiere en ambos casos características pasivas y activas. Un *estado pasivo* es simplemente el estado actual de todos los atributos de un objeto. El *estado activo* de un objeto indica el estado actual cuando éste entra en una transformación continua o proceso. Para forzar la transición de un objeto de un estado activo a otro debe ocurrir un evento (a veces, llamado *disparador*). Un componente de un modelo objeto–comportamiento es una representación simple de los estados activos de cada objeto y los eventos (disparadores) que producen los cambios entre estos estados activos.

En la representación gráfica de los estados activos de un objeto, las transiciones de un estado activo a otro se representan con flechas. Las etiquetas mostradas en cada flecha representan los eventos que disparan la transición.

El segundo tipo de representación de comportamiento para el AOO considera una representación de estados para el producto general o sistema. Esta representación abarca un modelo simple de traza de eventos [R$^+$91] que indica cómo los eventos

causan las transiciones de objeto a objeto y un diagrama de transición de estados que ilustra el comportamiento de cada objeto durante el procesamiento. Esta representación, llamada *traza de eventos (sucesos)*, es una versión abreviada del caso de uso.

Una vez que se ha desarrollado una traza completa de los eventos, todos aquellos que provoquen transiciones entre objetos del sistema pueden incluirse en un conjunto de eventos de entradas y eventos de salidas (desde un objeto). Esto puede representarse a partir de un *diagrama de flujo de eventos*.

### 3.1.3. Diseño Orientado a Objetos

El *Diseño Orientado a Objetos* (DOO) transforma el modelo de análisis creado usando el AOO en un modelo de diseño que sirve como un anteproyecto para la construcción del software. A diferencia de los métodos convencionales de diseño del software, el DOO constituye un tipo de diseño que logra un cierto número de niveles diferentes de modularidad. Las componentes principales del sistema están organizadas en "módulos" denominados subsistemas. Los datos y las operaciones que manipulan los datos están encapsulados en objetos, una forma modular que es el bloque de construcción de un sistema OO. En suma, el DOO debe describir la organización de datos específicos, de atributos y los detalles procedimentales de las operaciones individuales. Esta representación fragmentada de datos y algoritmos de un sistema OO colabora para lograr una modularidad general.

La naturaleza única del diseño orientado a objetos descansa en su capacidad de apoyarse en cuatro importantes conceptos de diseño de software: abstracción, ocultación de la información, independencia funcional y modularidad. Todos los métodos de diseño del software se afanan en mostrar estas importantes características, pero solamente el DOO aporta un mecanismo que le permite al diseñador alcanzar estas cuatro propiedades con menor complejidad y compromiso.

El trabajo del diseñador de software puede ser agotador. Gamma y sus colegas [G$^+$95] aportan una imagen razonablemente exacta del DOO cuando plantean que:

> El diseño del software orientado a objetos es duro y el diseño del software reusable orientado a objetos es aún más difícil. Se deben encontrar objetos pertinentes, distribuirlos en clases con la granularidad correcta, definir

interfaces de clases y jerarquías de herencia, y establecer relaciones clave entre ellas. El diseño debe ser específico al problema actual, pero también lo suficientemente general para resolver problemas y requisitos futuros. Se debe evitar también el rediseño, o al menos minimizarlo. Los diseñadores experimentados de software orientado a objetos dirán que es difícil, si no imposible, obtener un diseño flexible y reusable en la primera vez.

Para sistemas orientados a objetos se puede definir un diseño en pirámide. Como se muestra en la Figura 3.7, las cuatro capas del diseño OO son:

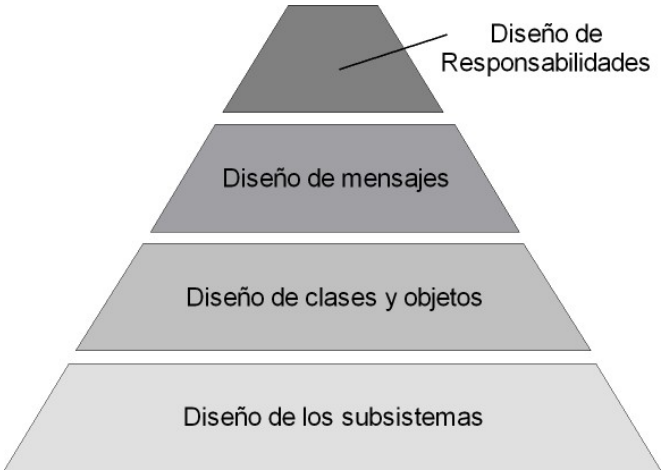


Figura 3.7: El diseño OO en pirámide

**La capa del subsistema.** Contiene una representación de cada uno de los subsistemas que le permiten al software conseguir los requisitos definidos por el cliente e implementar la infraestructura técnica que los soporta.

**La capa de clases y objetos.** Contiene las jerarquías de clases que permiten crear el sistema usando generalizaciones y especializaciones. Esta capa también contiene representaciones de diseño para cada objeto.

**La capa de mensajes.** Contiene los detalles que le permiten a cada objeto comunicarse con sus colaboradores. Esta capa establece las interfaces externas e internas para el sistema.

**La capa de responsabilidades.** Contiene las estructuras de datos y el diseño algorítmico para todos los atributos y operaciones de cada objeto.

### El Enfoque Convencional y el Enfoque OO

Como en el diseño del software convencional, el DOO aplica diseño de datos (cuando se representan atributos), diseño de interfaces (cuando se desarrolla un modelo de intercambio de mensajes), y diseño procedimental (en el diseño de operaciones). Sin embargo, el diseño arquitectónico es diferente. A diferencia del diseño arquitectónico derivado del uso de métodos de Ingeniería de Software convencionales, un diseño OO no exhibe una estructura de control jerárquica. De hecho, la "arquitectura" de diseño OO tiene más que ver con las colaboraciones entre objetos que con el flujo de control.

El diseño del subsistema se deriva considerando los requisitos generales del cliente (representados con casos de uso) y los eventos y estados observables externamente (el modelo objeto–comportamiento). Se establecen correspondencias para las clases y objetos a través de la descripción de atributos, operaciones y colaboraciones contenidas en el modelo CRC. El diseño de los mensajes se dirige por el modelo objeto–relación, y el diseño de las responsabilidades se obtiene a partir de los atributos, operaciones y colaboraciones descriptas en el modelo CRC.

### Asuntos del Diseño

Bertrand Meyer [Mey90] sugiere cinco criterios para juzgar la capacidad que posee un método de diseño para lograr la modularidad y los relaciona con el diseño orientado a objetos:

- *Descomposición Modular:* la facilidad con la cual un método de diseño ayuda al diseñador para descomponer un gran problema en subproblemas más sencillos de resolver.
- *Composición Modular:* el grado con el cual un método de diseño asegura que los componentes de un programa (módulos), una vez diseñados y construidos, pueden reusarse para crear otros sistemas.
- *Entendimiento Modular:* facilidad de comprensión de un componente de programa sin referencia a otra información o módulos.
- *Continuidad Modular:* la facilidad de hacer pequeños cambios en un programa y hacer que éstos se manifiesten por sí mismos en cambios correspondientes solamente en uno o pocos módulos más.

- *Protección Modular:* una característica arquitectónica que reducirá la propagación de efectos colaterales si ocurre un error en un módulo dado.

A partir de estos criterios Meyer [Mey90] sugiere cinco principios de diseño básicos para arquitecturas modulares: (1) unidades modulares lingüísticas; (2) pocas interfaces; (3) interfaces pequeñas (acoplamientos débiles); (4) interfaces explícitas; y (5) ocultamiento de la información.

Los módulos están definidos como *unidades modulares lingüísticas* cuando se "corresponden con unidades sintácticas en el lenguaje usado" [Mey90]. Esto es, el lenguaje de programación a usar debe ser capaz de soportar directamente la modularidad definida. Por ejemplo, si se definiera un *paquete* que contiene estructuras de datos y procedimientos, y se los identifica como una unidad simple, se necesitaría un lenguaje como Ada (u otro lenguaje orientado a objetos) para representar directamente este tipo de módulo en la sintaxis del lenguaje.

Para lograr un bajo acoplamiento debe minimizarse el número de interfaces entre módulos (pocas interfaces) y la cantidad de información que se mueve a través de una interfaz (interfaces pequeñas). Cada vez que se comuniquen los módulos, deben realizarlo de una manera obvia y directa (interfaces explícitas). Finalmente logramos el principio de ocultar la información cuando toda la información sobre un módulo está oculta al acceso exterior, a menos que la información se defina explícitamente como "información pública".

Los criterios y principios de diseño presentados en esta sección pueden aplicarse a cualquier metodología de diseño. Sin embargo, los enfoques orientados a objetos, gracias a sus características propias, alcanzan cada uno de los principios más eficientemente que otros enfoques, y resultan en arquitecturas modulares que nos permiten cumplir más eficientemente con todos los principios de modularidad.

### La Visión del DOO

A continuación se presentarán algunos de los métodos orientados a objetos más importantes, dando primero una visión general del método en sí, presentando luego la notación usada, mediante un ejemplo simple de cómo funciona una compañía, para concluir con una evaluación de cada uno de los métodos [Big00].

***Desarrollo Orientado a Objetos (OOD) / Booch***

Booch [Boo91] sugiere las siguientes etapas para analizar un sistema en preparación para diseñar una solución con una modalidad orientada a objetos:

1. Definir el problema
2. Desarrollar una estrategia informal para la realización mediante el software del dominio del problema del mundo real.
3. Formalizar la estrategia.

El problema se define en una descripción textual informal y concisa, y luego, de esta descripción puede obtenerse la información sobre los objetos y operaciones representados en el sistema. Generalmente, los objetos se representan con sustantivos y las operaciones con verbos. Las primeras dos etapas en realidad se llevan a cabo en el desarrollo durante la etapa de análisis de requerimientos del software, y no durante el diseño. Sin embargo, Booch ya había notado que el desarrollo orientado a objetos, como él lo describió, no es un método de ciclo de vida completo, sino que más bien se concentra en las etapas de diseño e implementación.

Para la formalización de la estrategia, Booch sugiere el siguiente orden de eventos:

- Identificar las clases y objetos en un nivel de abstracción dado.
- Identificar la semántica de estas clases y objetos.
- Identificar las relaciones entre estas clases y objetos.
- Implementar estas clases y objetos.

Booch se basa en estos principios para exponer su método. Destaca que éste no es sólo una simple secuencia de pasos a seguir, sino más bien un desarrollo iterativo e incremental a través del refinamiento de vistas complementarias (lógicas y físicas) de un sistema. Afirma que "el proceso de diseño orientado a objetos comienza con el hallazgo de las clases y objetos que forman el vocabulario del dominio de nuestro problema; y termina una vez que encontramos que no hay nuevas abstracciones y mecanismos primitivos o cuando las clases y objetos que ya hemos descubierto pueden ser implementados componiéndolos desde componentes existentes de software reusable".

En el método de diseño, Booch enfatiza la distinción entre la "vista lógica" de un sistema, en términos de clases y objetos, y la "vista física" de un sistema, en términos de módulos y procesos. También hace una distinción entre modelos estáticos y dinámicos de un sistema. El método que propone, sin embargo, está más dirigido a las descripciones estáticas del sistema, con menos soporte para las descripciones dinámicas.

Una de las principales fortalezas del método de Booch es la abundante notación disponible. Hay notaciones esquemáticas para producir diagramas de clases (estructura de clases - vista estática), diagramas de objetos (estructura de objetos - vista estática), diagramas de transición de estados (estructura de clase - vista dinámica), diagramas de secuencia (estructura de objetos - vista dinámica), diagramas de módulos (arquitectura de procesos), y diagramas de procesos (arquitectura de procesos).

Las notaciones para el modelado de clases y objetos usan comentarios o símbolos variantes (por ejemplo, diferentes tipos de flecha) para transmitir información detallada. Booch sugiere que en las etapas iniciales del diseño se puede usar un subconjunto de esas notaciones, y más tarde completar los detalles. También existe una forma textual para cada notación. El diagrama principal usado para describir la estructura de un sistema es el diagrama de clases, que muestra una vista estática de la estructura de clases en el diseño orientado a objetos. Los símbolos que se usan son los que aparecen en la Figura 3.8. La Figura 3.9 presenta un ejemplo de un diagrama de clases para una compañía.

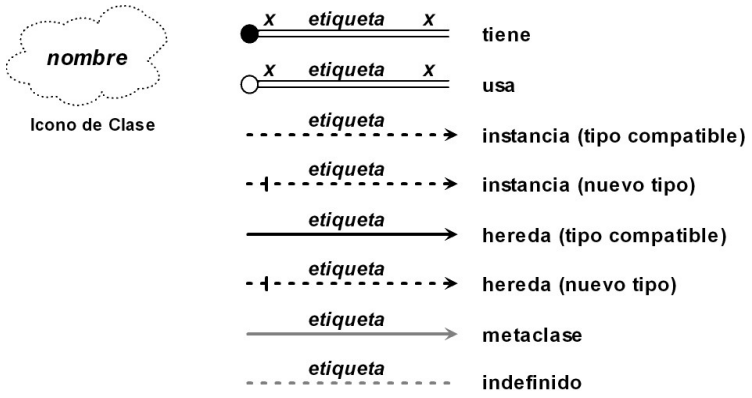


Figura 3.8: Una muestra de la notación del Diagrama de Clases de Booch

**Análisis del Método:** El enfoque de Booch es esencialmente pragmático. El método para diseño orientado a objetos nunca se desarrolla realmente dentro de un

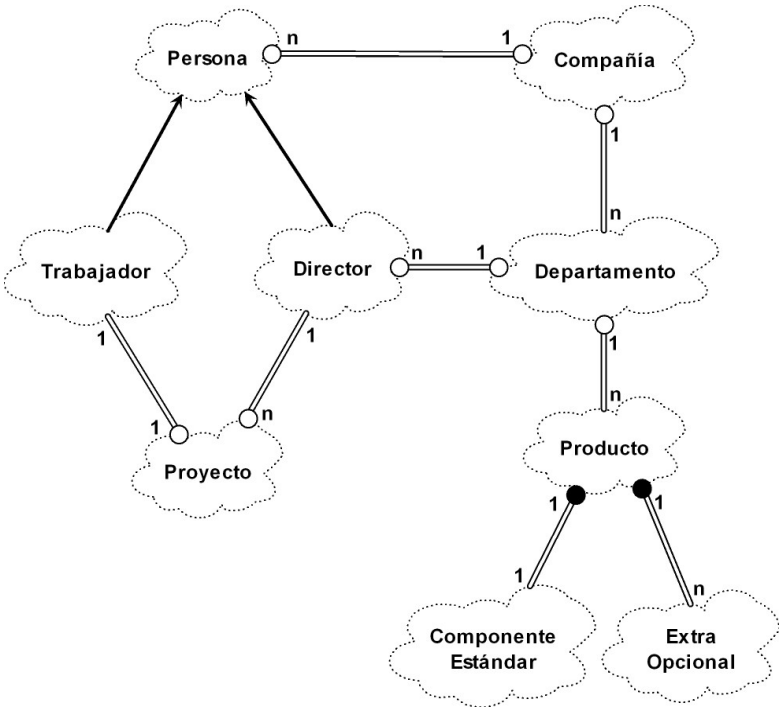


Figura 3.9: Un ejemplo de Diagrama de Clases de Booch

proceso, sino que más bien es una colección de técnicas, ideas formalizadas y heurísticas que pueden usarse cuando se desarrollan sistemas orientados a objetos. Booch da muchos buenos consejos en el campo del diseño orientado a objetos, pero generalmente no lo hace en forma de heurísticas explícitas. Esto puede considerarse tanto una fortaleza como una debilidad de este enfoque. Al evaluar este método, Ormsby [Orm91] considera que "lo que el método no dice es cómo debería decidir el diseñador si es posible o deseable seguir descomponiendo en una etapa cualquiera. Ni tampoco se proveen pautas para juzgar la calidad de una descomposición particular frente a otra (aunque esto es muy difícil)".

Sobre el enfoque orientado a objetos de Booch, Walker [Wal92] dice: "Un posible problema es sin duda la carencia de una partición y estratificación significativa de los diagramas en el método de Booch. Él emplea una variedad de diagramas para diferentes propósitos (diagramas de clases, diagramas de transición de estados, diagramas de relación/visibilidad de objetos y sincronización de mensajes, etc.), pero dentro de algunos diagramas parece querer cubrir demasiado del sistema (al menos, este parece ser el caso en los ejemplos y casos que él usa)." Walker continúa diciendo:

"Quizás la crítica a la estrategia de Booch podría encerrarse en la frase *amplitud antes que profundidad*. La notación de los diagramas de Booch han sacrificado detalle o profundidad para ganar extensión; esto puede ser útil en las etapas tempranas del proceso de diseño, pero debe ser capaz de adaptarse al creciente nivel de detalle. En parte, esto se realiza mediante las plantillas textuales, pero no están estructuradas de manera de relacionar los detalles a los diagramas de una manera favorable; de hecho, en algunas ocasiones, también carecen de la información esencial para la resolución de estructuras aceptables dentro de la aplicación".

***Diseño Orientado a Objetos Jerárquico (HOOD)***

Este método fue desarrollado por la Agencia Espacial Europea como un método de diseño y notación para Ada. Robinson [Rob92a] lo introduce de esta manera: "El enfoque de descomposición jerárquica *top-down* no es nuevo. Después de todo, este es el método usado con los diagramas de flujo de datos que comienzan con un diagrama de contexto que muestra todas las interfaces externas con un proceso central. Este proceso se descompone luego en otros procesos con flujos de datos y flujos de control interactuando entre ellos, con chequeos de consistencia entre niveles. Del mismo modo, el propósito de HOOD es desarrollar el diseño de un conjunto de objetos que juntos proveen la funcionalidad del programa".

El proceso principal en HOOD se llama Paso Básico de Diseño. Un Paso Básico de Diseño tiene como objetivo la identificación de los objetos hijos de un objeto padre dado, y de sus relaciones individuales con otros objetos existentes, o el refinamiento de un objeto terminal al nivel del código. Este proceso se basa en la identificación de objetos por medio de técnicas de diseño orientado a objetos.

En otro texto, Robinson va más allá en la definición del Paso Básico de Diseño [Rob92b]: "Un proceso del Paso Básico de Diseño se descompone a su vez en cuatro fases, definiendo así un micro ciclo de vida para un paso de diseño". Las fases se pueden resumir como sigue:

1. *Definición del problema.* Se enuncia el contexto del objeto a diseñar, con el objetivo de organizar y estructurar los datos de la fase de análisis de requerimientos. Esta es una oportunidad para proveer una prueba de completitud sobre los requerimientos y su rastreabilidad al diseño.

   *a*) Declaración del problema – el diseñador enuncia el problema en oraciones

correctas que proveen:

- Una definición clara y precisa del problema
- El contexto del sistema a diseñar.

*b*) Análisis y estructuración de datos de los requerimientos – el diseñador recoge y analiza toda la información relevante al problema, incluyendo el entorno del sistema a diseñar.

2. *Desarrollo de la estrategia de solución.* Se describe la solución preliminar del problema descripto anteriormente, en términos de objetos con un alto nivel de abstracción.

3. *Formalización de la estrategia.* Se definen los objetos y sus operaciones asociadas. Se produce un diagrama HOOD de la solución de diseño propuesta, permitiendo la fácil visualización de los conceptos y su posterior formalización. Hay cinco subfases dentro de la formalización de la estrategia:

   *a*) Identificación de objetos.

   *b*) Identificación de operaciones.

   *c*) Agrupamiento de objetos y operaciones (tabla de operación de objetos).

   *d*) Descripción gráfica.

   *e*) Justificación de las decisiones de diseño.

4. *Formalización de la solución.* La solución se formaliza mediante:

   - Una definición formal de las interfaces de objetos provistas
   - Una descripción formal de los objetos y las estructuras de control.

El principal diagrama usado para describir la estructura de un sistema es el diagrama de objetos HOOD, que muestra una vista estática de la estructura en el diseño jerárquico orientado a objetos. Los símbolos usados son los que aparecen en la Figura 3.10. La Figura 3.11 presenta un ejemplo de un diagrama HOOD para una compañía.

**Análisis del Método:** HOOD está fuertemente orientado a una implementación en Ada. Obviamente, esto es ideal para desarrolladores de Ada, pero puede considerarse que limita su utilidad para el entorno de cualquier otro lenguaje de programación.

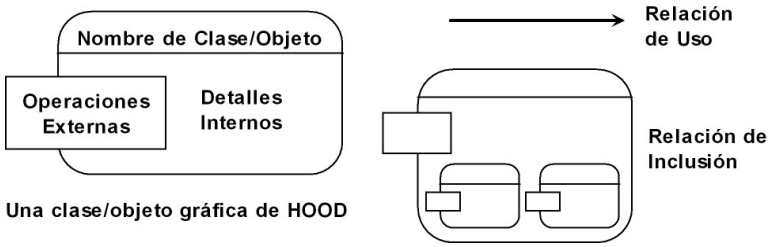


Figura 3.10: Una muestra de la notación de un Diagrama HOOD

Jacobson *et al.* [Jac92] dicen: "El método da el paso básico de diseño, pero no brinda ninguna ayuda para encontrar la estructura de objetos apropiada. En realidad HOOD sí da un fuerte soporte para estructuras de almacenamiento, pero no para otras estructuras como las de uso o herencia".

Una de las carencias principales de HOOD es su falta de soporte para algunas de las técnicas disponibles más orientadas a objetos. El soporte para herencia es muy pobre, como se evidencia por la falta de una notación para representarla gráficamente. El método quizás sea más *basado* en objetos que verdaderamente orientado a objetos.

***Técnica de Modelado de Objetos (OMT) / Rumbaugh***

La Técnica de Modelado de Objetos [R+91] provee tres conjuntos de conceptos que a su vez brindan tres vistas diferentes del sistema. Hay un método que conduce a tres modelos del sistema correspondientes a estas tres vistas. Los modelos se definen inicialmente, luego se refinan a medida que progresan las fases del método. Los tres modelos son:

- El *modelo de objetos*, que describe la estructura estática de los objetos de un sistema y sus relaciones. Los conceptos principales son: Clase, Atributo, Operación, Herencia, Asociación (es decir, relación), Agregación.

- El *modelo dinámico*, que describe los aspectos del sistema que cambian a través del tiempo. Este modelo se usa para especificar e implementar los aspectos de control de un sistema. Los conceptos principales son: Estado, Sub/súper estado, Evento, Acción, Actividad.

- El *modelo funcional*, que describe las transformaciones de los valores de los datos dentro de un sistema. Los conceptos principales son: Proceso, Almacenamiento de datos, Flujo de datos, Flujo de control, Actor (fuente / sumidero).

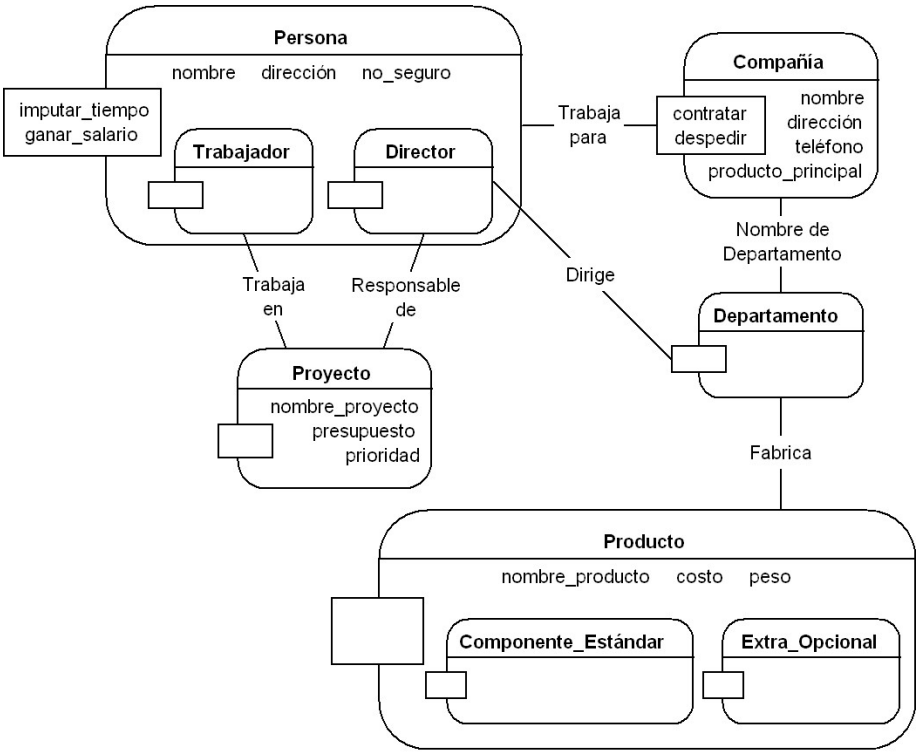


Figura 3.11: Un ejemplo de Diagrama HOOD

El método se divide en cuatro fases, que son etapas del proceso de desarrollo:

1. *Análisis* – la construcción de un modelo de la situación del mundo real, en base a una enunciación del problema o a requerimientos de usuario. Los entregables de la etapa de análisis son: Enunciación del Problema, Modelo de Objetos (Diagrama del Modelo de Objetos + Diccionario de Datos), Modelo Dinámico (Diagramas de Estados + Diagrama de Flujo de Eventos Global), y Modelo Funcional (Diagrama de Flujo de Datos + restricciones).

2. *Diseño del Sistema* – la partición del sistema objetivo en subsistemas, en base a una combinación de reconocimiento del dominio del problema y la arquitectura propuesta del sistema objetivo (dominio de solución). El entregable de la etapa de diseño del sistema es el Documento de Diseño del Sistema (arquitectura básica del sistema y decisiones estratégicas de alto nivel).

3. *Diseño de Objetos* – construcción de un diseño, en base al modelo de análisis enriquecido con detalles de implementación, incluyendo las clases de infraestruc-

tura del dominio de las computadoras. Los entregables de la etapa de diseño de objetos son: Modelo Detallado de Objetos, Modelo Dinámico Detallado, y Modelo Funcional Detallado.

4. Implementación – traducción del diseño a un lenguaje o a una instanciación de hardware particular, con destacado énfasis en la rastreabilidad y manteniendo la flexibilidad y la extensibilidad.

El diagrama principal usado para describir la estructura de un sistema es el modelo de objetos, que brinda un modelo de la estructura de clases especificada en el DOO. Los símbolos que se usan se muestran en la Figura 3.12. La Figura 3.13 presenta un modelo de objetos para el mismo ejemplo del caso anterior.

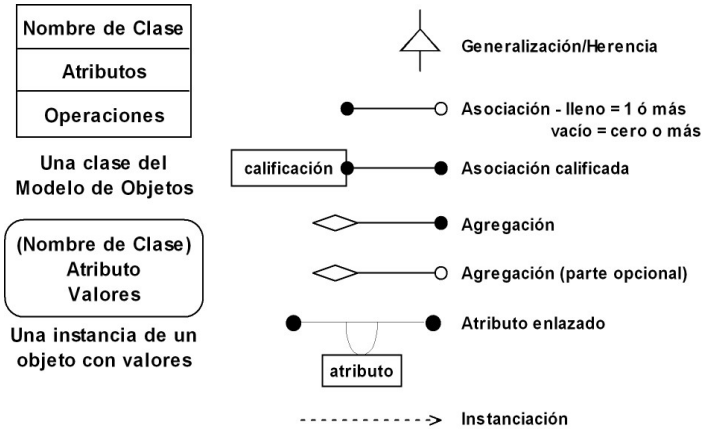


Figura 3.12: Una muestra de la notación OMT

**Análisis del Método:** El OMT es quizás uno de los métodos de DOO más desarrollado, tanto en términos de la notación que usa, como de los procesos que se recomiendan para desarrollar un sistema orientado a objetos. Al describir la técnica, Rumbaugh *et al.* [R$^+$91] aseguran que el método intenta mostrar cómo usar los conceptos orientados a objetos durante todo el ciclo de vida de desarrollo del software. La inclusión de secciones que discuten el análisis, diseño, e implementación tanto en lenguajes que son orientados a objetos como en aquellos que no lo son ayuda a validar esta afirmación.

Potencialmente, el concepto presentado en el OMT de usar tres vistas para representar un sistema es muy poderoso, pero también puede considerarse como muy

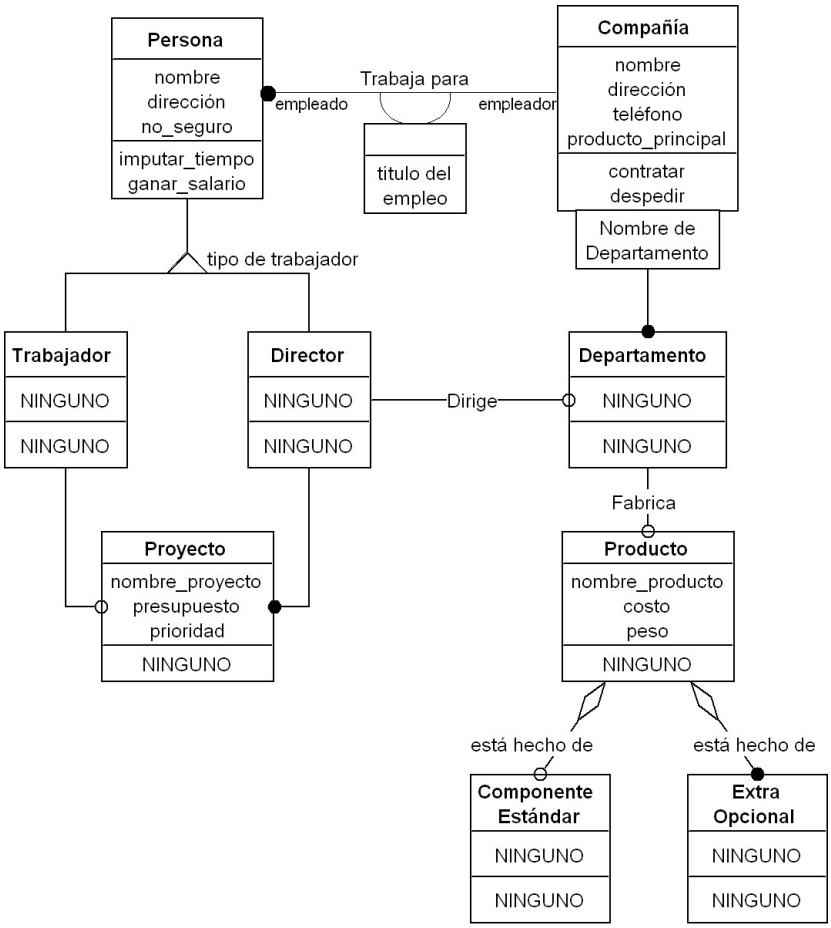


Figura 3.13: Ejemplo de un Modelo de Objetos OMT para una compañía

complejo. No está claro si las vistas tienen que ser desarrolladas de manera independiente unas de otras, o si el conocimiento de un modelo debería usarse para influenciar la construcción de los demás. Parecería lógico tener que asegurar que los conceptos de las diferentes vistas estén adecuadamente definidos e interrelacionados, para que no haya confusiones en la etapa de diseño del desarrollo, donde esos modelos deben ser integrados.

Walker [Wal92] nota deficiencias del método OMT en dos áreas principales. Primero, "no hay un intento serio de abordar el reuso sistemático de software o de componentes de diseño". Segundo, "la dirección del proceso de desarrollo del software, incluyendo el establecimiento de métricas adecuadas para la medición de progreso y calidad, está casi totalmente descuidada". Sin embargo, considera que, en términos de un enfoque técnico para el desarrollo de software orientado a objetos, "Rumbaugh *et al.* han establecido claramente el estado actual del arte".

***Diseño Conducido por Responsabilidades (RDD) / Clase–Responsabilidad–Colaboración (CRC) / Wirfs-Brock***

En el Diseño Conducido por Responsabilidades [WB+90], un modelo se desarrolla desde la especificación de requerimientos mediante la extracción de los sustantivos y verbos de la especificación. Esto provee las bases para la implementación real. Se cubren todos los conceptos básicos de la orientación a objetos.

En RDD, para cada clase se definen diferentes responsabilidades que especifican los roles de los objetos y sus acciones. Para satisfacer estas responsabilidades, las clases necesitan colaborar entre sí. Se definen las colaboraciones para mostrar cómo van a interactuar los objetos. Las responsabilidades son luego agrupadas en contratos que definen un conjunto de solicitudes que los objetos de la clase pueden soportar. Estos contratos son nuevamente refinados en protocolos, que muestran la signatura específica de cada operación.

Se introducen los subsistemas para agrupar un número de clases y subsistemas de menor nivel con el objetivo de abstraer una cierta funcionalidad. Los subsistemas también tienen contratos que deben ser soportados por alguna clase del subsistema. El método RDD consta de dos fases principales:

1. La fase exploratoria, que consiste en:
    - Clases
        - Extraer frases nominales de la especificación y conformar una lista.
        - Identificar clases candidatas de las frases nominales.
        - Identificar candidatas a superclases abstractas.
        - Usar categorías para buscar clases ausentes.
        - Escribir una corta declaración del propósito de cada clase.
    - Responsabilidades
        - Detectar responsabilidades.
        - Asignar las responsabilidades a las clases.
        - Detectar otras responsabilidades observando las relaciones entre clases.
    - Colaboraciones
        - Encontrar y enumerar colaboraciones examinando las responsabilidades asociadas a las clases.

- Identificar colaboraciones adicionales observando las relaciones entre clases.
- Descartar las clases que no participan en ninguna colaboración (como clientes o servidoras).

2. La fase de análisis, que consiste en:

   - Jerarquías
     - Dibujar grafos de jerarquía de herencia.
     - Identificar qué clases son abstractas/concretas.
     - Dibujar diagramas de Venn que muestren cómo las responsabilidades son compartidas por las clases.
     - Refinar la jerarquía de clases, chequeando la asignación de las responsabilidades.
     - Definir cómo se agrupan las responsabilidades en contratos, y qué clases soportan cada contrato.
   - Subsistemas
     - Dibujar un grafo completo de colaboraciones para el sistema.
     - Identificar posibles subsistemas.
     - Simplificar las colaboraciones entre y dentro de los subsistemas.
   - Protocolos
     - Definir protocolos para cada clase, refinando las responsabilidades en conjuntos de signaturas de métodos.
     - Escribir una especificación para cada clase.
     - Escribir una especificación para cada subsistema.
     - Escribir una especificación para cada contrato.

El principal diagrama usado para describir la estructura de un sistema es el grafo de colaboraciones, que representa los subsistemas, clases, contratos y colaboraciones en el sistema. La Figura 3.14 muestra los símbolos más comúnmente usados, mientras que en la Figura 3.15 se presenta un grafo de colaboraciones para el mismo ejemplo de la compañía.

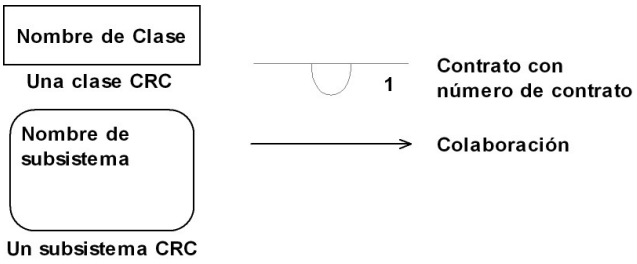


Figura 3.14: Una muestra de la notación CRC

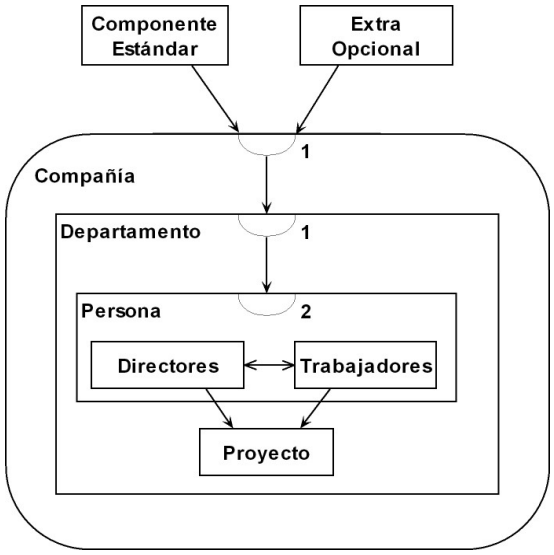


Figura 3.15: Un ejemplo de un Grafo de Colaboraciones para una compañía

**Análisis del Método:** El método RDD tiende a usar técnicas y lineamientos informales para progresar sobre un diseño apropiado. Confía fuertemente en las habilidades del diseñador para encontrar las clases y sus propiedades. Una parte importante de la estrategia de diseño es el análisis textual de un documento de especificación de requerimientos para identificar clases y responsabilidades. Las tarjetas CRC, que ya se describieron en la Sección 3.1.2, son útiles para capturar especificaciones de clases y subsistemas. De acuerdo a Wirfs-Brock *et al.* [WB$^+$90], las tarjetas índice son ideales para la tarea porque son compactas y fáciles de manipular, modificar y descartar. También pueden ser fácilmente distribuidas sobre una mesa, y reacomodadas para explorar nuevas perspectivas, o dispuestas de manera tal de dejar espacios vacíos donde parece haber clases faltantes.

Este tipo de simplicidad puede considerarse tanto una fortaleza como una de-

bilidad. Hace que el método de diseño sea mucho más manejable notacional e intelectualmente; sin embargo, hay una necesidad muy fuerte de que el documento de especificación de requerimientos (u otras fuentes de información) sea lo suficientemente claro, preciso y comprensible como para expresar un diseño usando los conceptos recomendados.

RDD sólo considera la actividad de diseño, no hay ponderación del análisis del dominio o del problema, o de la captura de requerimientos. El concepto de clase es central –todos los demás conceptos están para describir las clases y sus dependencias. La actividad de diseño produce lo que en realidad es un único modelo en una única notación. No hay otros modelos, como modelos del comportamiento dinámico. La notación tampoco es lo bastante expresiva como para mostrar el diseño capturado por el proceso.

Las heurísticas de este método de diseño son explícitas y generalmente bien descriptas. Gossain [Gos91] considera que Wirfs-Brock *et al.* proveen una introducción sumamente útil a los conceptos centrales del diseño orientado a objetos, pero que también pierden la visión de conjunto. Por último, no cree que este enfoque de diseño pueda escalarse correctamente a sistemas grandes.

***Análisis Orientado a Objetos (OOA) / Coad y Yourdon***

OOA usa principios estructurales básicos, y los reúne con un punto de vista orientado a objetos [CY90, CY91]. El método consiste en cinco etapas:

1. Encontrar Clases y Objetos – especifica cómo se deberían encontrar las clases y objetos. El primer enfoque está dado comenzando con el dominio de la aplicación e identificando las clases y objetos que forman la base de la aplicación completa, y analizando luego las responsabilidades del sistema en este dominio.

2. Identificar Estructuras – esto se hace de dos formas diferentes. Primero la estructura de generalización–especialización, que captura la jerarquía entre las clases especificadas. Segundo, la estructura todo–partes, que se usa para modelar cómo un objeto es parte de otro objeto, y cómo los objetos se componen en categorías más grandes.

3. Definir Temas – esto se hace dividiendo el modelo de Clases y Objetos en unidades mayores. Los temas son grupos de clases y objetos. Se pueden usar las estructuras identificadas anteriormente.

4. Definir Atributos – esto se hace identificando información y asociaciones para cada instancia. Esto implica identificar los atributos necesarios para caracterizar cada objeto. Los atributos identificados se ubican en el nivel correcto de la jerarquía de herencias.

5. Definir Servicios – definir las operaciones de las clases. Esto se hace identificando los estados de los objetos y definiendo servicios para acceder y alterar ese estado.

El resultado final de la etapa de análisis es un modelo del dominio del problema en términos de cinco capas:

- Capa de Temas
- Capa de Clases y Objetos
- Capa de Estructuras (es decir, herencias y relaciones)
- Capa de Atributos
- Capa de Servicios

Identificar los elementos de cada capa constituye una actividad dentro del método. El orden de abstracción no es importante, aunque generalmente es más fácil ir desde niveles altos de abstracción hacia los niveles más bajos.

Un modelo de diseño orientado a objetos consta de los siguientes componentes:

- Componente del Dominio del Problema (PDC) – el resultado del análisis orientado a objetos se pone directamente en esta capa.
- Componente de Interacción Humana (HIC) – éste implica actividades tales como: clasificar usuarios humanos, describir escenarios de tareas, diseñar la jerarquía de comandos, diseñar la interacción detallada, hacer un prototipo de la Interfaz Humano–Computadora, definir clases HIC.
- Componente de Gestión de Tareas (TMC) – este componente consiste en la identificación de tareas (procesos), los servicios que ellas proveen, la prioridad de las tareas, si el proceso es conducido por eventos o por reloj, y cómo se comunica (con otros procesos y con el mundo exterior).

- Componente de Gestión de Datos (DMC) – este componente depende en gran medida de la tecnología de almacenamiento disponible, y de la persistencia de los datos requerida.

El diagrama principal usado para describir la estructura de un sistema es el modelo AOO, que brinda una vista estática de la estructura de clases en el diseño orientado a objetos. Los símbolos usados son los que se muestran en la Figura 3.16. La Figura 3.17 ilustra un ejemplo de un modelo OOA de Coad y Yourdon para el mismo ejemplo de la compañía.

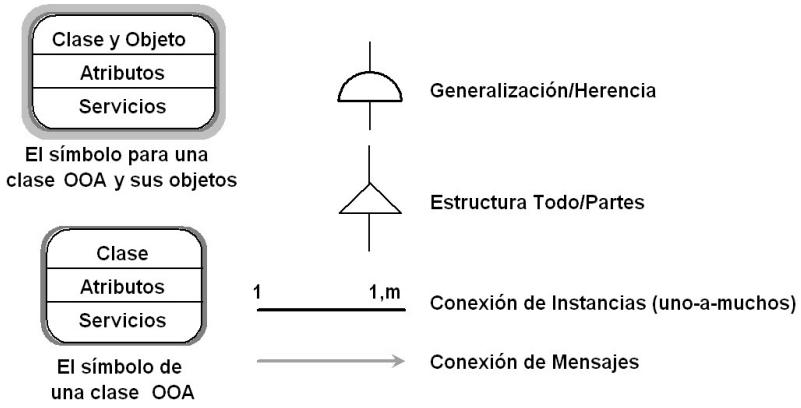


Figura 3.16: Una muestra de la notación OOA

**Análisis del Método:** El método orientado a objetos de Coad y Yourdon da la impresión de haber sido rápidamente adaptado de sus otros trabajos justo a tiempo para subirse al tren de la orientación a objetos. Bilow [Bil91] nota que algunas porciones de su exposición del método están "excesivamente detalladas para lectores encargados de la dirección, mientras que otras secciones están demasiado generales para diseñadores e implementadores".

Los cuatro componentes del enfoque de diseño no se presentan con suficiente detalle para ser puestos directamente en práctica. Sin embargo, su enfoque al análisis y diseño orientado a objetos es fácil de comprender, y su método se transmite en un nivel que evita algunas de las difíciles complejidades de la orientación a objetos formalizada. Como tal, brinda una buena introducción a la orientación a objetos. Sin embargo, hay suficientes defectos en el método como para evitarlo para sistemas grandes o altamente complejos.

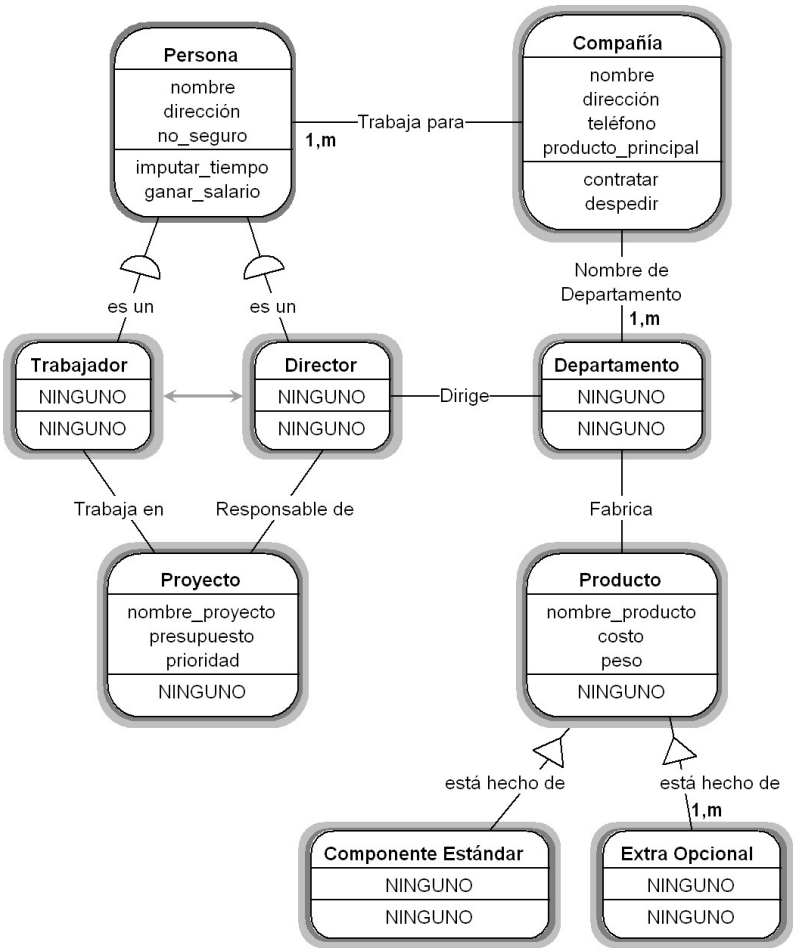


Figura 3.17: Ejemplo del modelo OOA de Coad y Yourdon para la compañía

***Análisis Estructurado Orientado a Objetos (OOSA) / Lenguaje de Diseño Orientado a Objetos (OODL) / Shlaer/Mellor***

En este método [SM92], el sistema de software es inicialmente dividido en dominios, que luego pueden ser desmembrados en subsistemas para ser analizados en profundidad. El método provee una cobertura integral para las etapas de análisis, diseño e implementación del ciclo de vida de desarrollo de software. El proceso puede describirse en las etapas siguientes:

1. Dividir el sistema en dominios – el dominio de aplicación se define como el contenido de interés para el usuario final del sistema. El dominio de servicios pueden incluir la interfaz de usuario y otras interfaces externas del sistema. El dominio de la arquitectura cubre el hardware y software que se usará para

construir y ejecutar el sistema deseado. El dominio de implementación incluirá el sistema operativo y el lenguaje de programación que se usará para construir el sistema.

2. Analizar el dominio de aplicación – el dominio de aplicación se analiza usando el conjunto integrado de modelos de análisis orientado a objetos de Shlaer/Mellor.

3. Verificar el análisis mediante la simulación – los modelos OOSA proveen un método formal para verificar el comportamiento especificado del sistema mediante la simulación de la ejecución de los modelos. En estas simulaciones se usan los procesos, datos y secuencia de procesos para verificar la ejecución de los modelos definidos.

4. Extraer los requerimientos para el dominio de servicios – se considera cada dominio de servicio en términos de los requerimientos que cada uno de los otros dominios le hacen. Esto permite que, para el próximo paso, se identifique el propósito de cada dominio.

5. Analizar el dominio de servicios – el dominio de servicios se analiza usando los modelos de OOSA, mediante el mismo proceso usado en el paso 2.

6. Especificar los componentes del dominio de la arquitectura – el dominio de la arquitectura se especifica usando OOSA, pero se puede diseñar usando cualquier método de diseño que se estime apropiado. Si se requiere un diseño orientado a objetos, se recomienda el Lenguaje de Diseño Orientado a Objetos (OODL). OODL usa cuatro tipos de diagramas interrelacionados por un esquema de capas para representar el diseño de un programa, biblioteca o ambiente orientado a objetos: diagramas de Herencia, diagramas de Dependencia, diagramas de Clases, y gráficos de Estructura de Clases

7. Construir los componentes de la arquitectura – en el dominio de la arquitectura existen dos componentes: Mecanismos, que representan las capacidades específicas de la arquitectura que se deben proveer para realizar el sistema, y Estructuras, que representan una fórmula para traducir los modelos OOSA de los dominios del cliente.

8. Traducir los modelos de OOSA de cada dominio usando los componentes de arquitectura – los detalles del paso final dependen en gran medida del diseño elegido para el sistema, y de los componentes de arquitectura creados.

Hay varios diagramas que se usan para describir la estructura de un sistema, a saber, el diagrama de herencia, los diagramas de dependencias y el diagrama de clases. Por conveniencia, estos se han combinado para mostrar el diseño orientado a objetos del ejemplo. Los símbolos que se usan en la notación aparecen en la Figura 3.18, mientras que la Figura 3.19 muestra el ejemplo en sí.

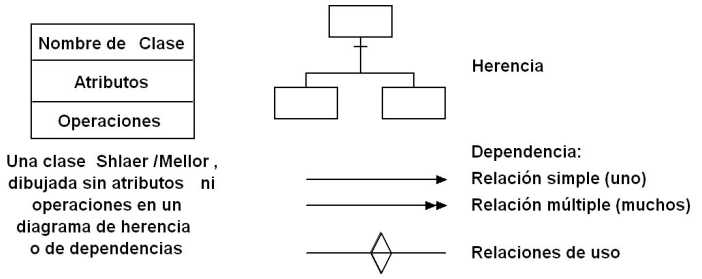


Figura 3.18: Una muestra de la notación OODL

**Análisis del Método:** El método de análisis y diseño orientado a objetos de Shlaer/Mellor fue uno de los primeros intentos de definir un método estructurado para orientación a objetos. Originalmente era más basado en objetos que verdaderamente orientado a objetos, pero ha sido refinado desde su concepción inicial para incluir muchos de los elementos clave de la orientación a objetos.

La simulación del modelo OOSA está bien definida dentro del método, y provee un medio para probar la validez del sistema capturado por los modelos. Debido a la naturaleza estructurada de la simulación, gran parte del proceso puede automatizarse con éxito. El proceso de completar las estructuras del dominio de la arquitectura también puede automatizarse parcialmente, haciendo altamente reusables las plantillas construidas como resultado de este método.

### *Proceso Unificado de Desarrollo de Software / Jacobson/Booch/Rumbaugh*

Este método [JBR99] fue desarrollado en un esfuerzo conjunto para incluir las bondades de los métodos desarrollados individualmente por cada uno de los autores.

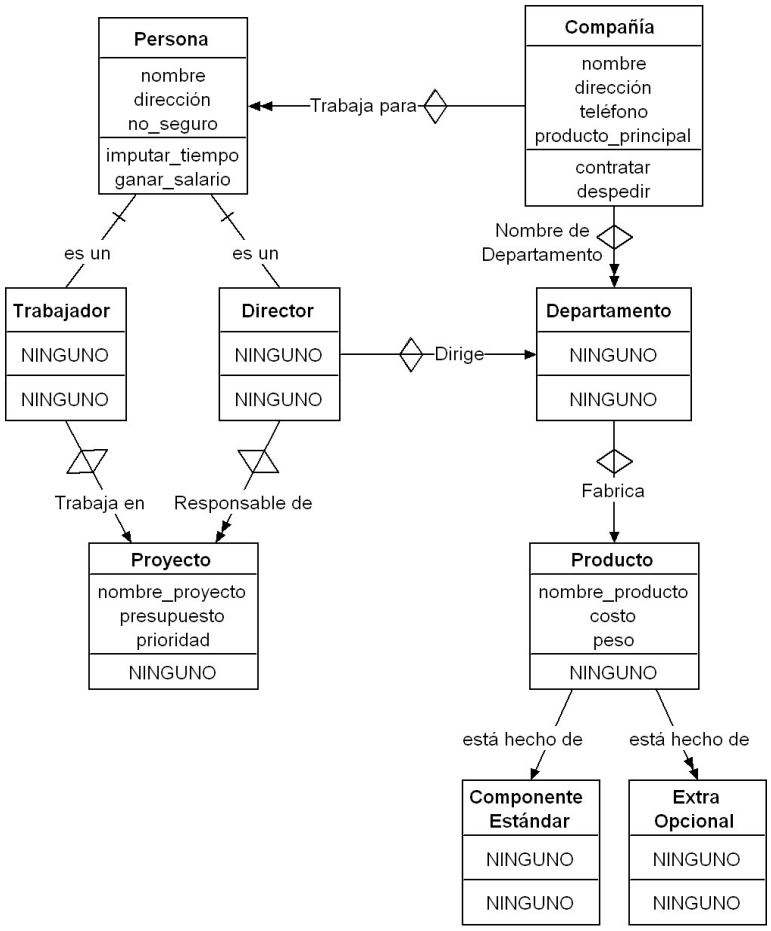


Figura 3.19: Ejemplo de un diagrama OODL simplificado para la compañía

También desarrollaron un Lenguaje Unificado de Modelado (UML) para la especificación de los modelos. El Proceso Unificado se verá en detalle en la Sección 3.2, mientras que en la Sección 3.3 se examinarán las principales características de UML.

### El Proceso de Diseño del Sistema

Aunque cierto número de autores sugiere modelos de procesos para el diseño de sistemas OO, la secuencia de actividades propuesta por Rumbaugh y sus colegas [R$^+$91] es uno de los tratamientos más definitivos a este tema. Dicha secuencia consta de las siguientes etapas:

- Dividir el modelo de análisis en subsistemas.

  En el diseño de sistemas OO se divide el modelo de análisis para definir colecciones cohesivas de clases, relaciones y comportamientos. Estos elementos del

diseño, que en general comparten alguna propiedad en común, se empaquetan como un subsistema. Los subsistemas se caracterizan por sus responsabilidades; esto es, un subsistema puede identificarse por los servicios que realiza. Un *servicio* es una colección de operaciones que realizan una función específica.

- Identificar la concurrencia dictada por el problema.
  Los aspectos dinámicos del modelo de objeto–comportamiento nos brindan una indicación de concurrencia entre objetos (o entre subsistemas). Si los objetos (o subsistemas) no están activos al mismo tiempo, no existe necesidad de procesamiento concurrente. Por otra parte, los objetos (o subsistemas) se toman como concurrentes si deben actuar sobre eventos asincrónicamente y al mismo tiempo.

- Asignar subsistemas a procesadores y tareas.
  Cuando los subsistemas son concurrentes, existen dos opciones de asignación:

  - Asignar cada subsistema a un procesador independiente.
  - Asignar los subsistemas al mismo procesador y ofrecer soporte de concurrencia a través de las capacidades del sistema operativo.

- Elegir una estrategia básica para la implementación de la gestión de datos.
  La gestión de datos abarca dos áreas distintas: (1) la gestión de datos críticos para la propia aplicación, y (2) la creación de una infraestructura para el almacenamiento y recuperación de objetos. En general, la gestión de datos se diseña por capas. La idea es aislar los requisitos de bajo nivel para la gestión de datos de los de alto nivel para la gestión de los atributos del sistema.

- Identificar los recursos globales y los mecanismos de control necesarios para acceder a ellos.
  Existe una variedad de recursos diferentes disponibles para un sistema o producto OO, y en muchas instancias, los subsistemas compiten por estos recursos al mismo tiempo. El ingeniero de software debe diseñar un mecanismo de control sobre ellos, por ejemplo haciendo que cada recurso sea propiedad de un "objeto guardián", que actúa controlando los accesos a él y moderando las solicitudes en conflicto sobre él.

- Diseñar un mecanismo de control apropiado para el sistema.
- Considerar cómo manipular las condiciones límite.
- Revisar y considerar los intercambios.
  Una vez que se ha especificado cada subsistema, es necesario definir las colaboraciones que existen entre ellos. El modelo que usamos para la colaboración objeto-a-objeto puede extenderse a subsistemas de forma general. Debemos especificar el contrato que existe entre los subsistemas. Un contrato brinda indicaciones acerca de las maneras en que un subsistema puede interactuar con otro.

**El Proceso de Diseño de Objetos**

El diseño de sistemas OO puede verse como el plano arquitectónico de una casa. Este plano especifica el propósito de cada habitación y los mecanismos que conectan las habitaciones unas con otras y el entorno exterior. Es ahora el momento de dar los detalles necesarios para construir cada habitación.

En el contexto del DOO, el diseño de objetos se centra en las "habitaciones". Debemos desarrollar un diseño detallado de los atributos y operaciones que incluye cada clase, y una especificación completa de los mensajes que conectan la clase con sus colaboradores.

- *Descripciones de Objeto*
  Una descripción de un objeto (una instancia de una clase o subclase) puede tomar una de estas formas [GR83a]:
  1. Una *descripción del protocolo* que establece la interfaz de un objeto definiendo cada mensaje que el objeto puede recibir y la correspondiente operación que el mismo ejecuta al recibir el mensaje, y
  2. Una *descripción de la implementación* que muestra detalles de ella para cada operación implicada por un mensaje que se pasa al objeto. Los detalles de implementación incluyen información acerca de la parte privada del objeto.
- *Diseño de Algoritmos y Estructuras de Datos*
  Los algoritmos y estructuras de datos de un sistema OO se diseñan usando un

enfoque bastante similar al del diseño de datos y los procedimientos examinados para la Ingeniería de Software convencional.

Los algoritmos se crean para implementar la especificación de cada operación. En muchos casos, el algoritmo es una simple secuencia computacional o procedimental que puede implementarse como un módulo del software autocontenido. Sin embargo, si la especificación de la operación es compleja, pudiera ser necesario la modularización de la operación.

Las estructuras de datos se diseñan concurrentemente con los algoritmos. Debido a que las operaciones manipulan invariablemente los atributos de una clase, el diseño de las estructuras de datos que mejor reflejan los atributos tendrá una fuerte influencia sobre el diseño algorítmico de las operaciones correspondientes.

- *Componentes de Programas e Interfaces*

  Un aspecto importante de la calidad del diseño del software es la *modularidad*, esto es, la especificación de *componentes del programa* (módulos) que se combinan para formar un programa completo. El enfoque orientado a objetos define al objeto como un componente del programa que está auto-enlazado con otros componentes (por ejemplo: datos privados, operaciones).

  Pero definir objetos y operaciones no es suficiente. Durante el diseño, debemos identificar las *interfaces* que existen entre objetos y la estructura general (considerada desde un punto de vista arquitectónico) de los objetos.

### Patrones de Diseño

Los mejores diseñadores en cualquier campo poseen una habilidad innata para ver patrones que caracterizan un problema, y los patrones correspondientes que pueden combinarse para crear una solución. Gamma y sus colegas [G$^+$95] expresan:

> Usted encontrará patrones recurrentes de clases y objetos en comunicación. Estos patrones resuelven problemas de diseño específicos y hacen el diseño orientado a objetos más flexible, elegante y en última instancia reusable. Ayudan a los diseñadores a reutilizar diseños de éxito basando el nuevo diseño en experiencias anteriores. Un diseñador familiarizado con tales patrones puede aplicarlos de manera inmediata a problemas de diseño sin tener que redescubrirlos.

***Descripción de un Patrón de Diseño***

Todos los patrones de diseño pueden describirse a través de la especificación de cuatro piezas de información [G+95]

- El nombre del patrón.
- El problema al cual se aplica generalmente el patrón.
- Las características del patrón de diseño.
- Las consecuencias de la aplicación del patrón de diseño.

El nombre del patrón de diseño es una abstracción que aporta un significado acerca de su aplicabilidad y objetivos. La descripción del problema indica el entorno y las condiciones que deben existir para hacer aplicable el patrón de diseño. Las características del patrón indican los atributos del diseño que deben ajustarse para permitirle al patrón acomodarse a una variedad de problemas. Estos atributos representan características del diseño según las cuales puede buscarse el patrón apropiado. Finalmente, las consecuencias asociadas con el uso del patrón de diseño aportan una indicación de las ramificaciones de las decisiones de diseño.

***Clasificación de los Patrones de Diseño***

Los patrones de diseño se pueden clasificar teniendo en cuenta dos criterios. El primero de ellos es el *propósito*, es decir, lo que hace el patrón. De acuerdo con este criterio, existen tres tipos de patrones:

- **De Creación:** se aplican a procesos de creación de objetos.
- **De Estructura:** describen composición de clases y objetos.
- **De Comportamiento:** caracterizan la interacción entre objetos y sus responsabilidades.

El segundo criterio de clasificación tiene en cuenta si el patrón se aplica principalmente a clases o a objetos.

- **Clases:** patrones que describen las relaciones estáticas entre clases y subclases.
- **Objetos:** patrones que describen objetos y sus relaciones dinámicas.

***Uso de Patrones en el Diseño***

En un sistema orientado a objetos, los patrones de diseño pueden usarse aplicando dos mecanismos diferentes: herencia y composición. A través del uso de la *herencia* un patrón de diseño existente se convierte en una plantilla para una subclase nueva. Los atributos y operaciones que existen en el patrón pasan a ser parte de la clase.

La *composición* es un concepto que nos lleva al de objetos agregados. Esto es, un problema puede necesitar objetos que poseen una funcionalidad compleja. El objeto complejo puede ensamblarse a partir de la selección de un conjunto de patrones de diseño y la composición del objeto (o subsistema) seleccionado. Cada patrón de diseño se trata como una caja negra, y la comunicación entre ellos ocurre solamente a través de interfaces bien definidas.

Gamma y sus colegas [G$^+$95] sugieren que la composición de objetos debe ser favorecida por encima de la herencia cuando existen ambas opciones. La composición favorece pequeñas jerarquías de clases y objetos que permanecen centrados en un objetivo. La composición usa patrones de diseño existentes (componentes reusables) de forma inalterada.

***Un ejemplo: el patrón Adapter***

- **Clasificación:** Clase–Objeto, estructural.
- **Objetivo:** convertir la interfaz de una clase en otra interfaz que el cliente necesita.
- **Aplicabilidad:**
  - se quiere utilizar una clase existente, pero su interfaz no concuerda con lo que se necesita
  - se quiere crear una clase reusable que coopere con clases no relacionadas, o que no necesariamente tienen interfaces compatibles.
  - en el caso del patrón de objetos, se necesita usar varias subclases existentes, pero no es práctico adaptar las interfaces de cada una.
- **Participantes:**
  - *ClaseAdaptada*: define la interfaz específica del dominio que usa el cliente.

- *Cliente*: interactúa con la interfaz de la clase anterior.
- *ClaseAAdaptar*: representa una clase existente con interfaz incompatible.
- *Adaptador*: clase intermedia que efectúa la adaptación.

- **Estructura:** En la Figura 3.20 se muestra la estructura del adaptador de clases. Éste utiliza herencia múltiple para adaptar una interfaz a otra.

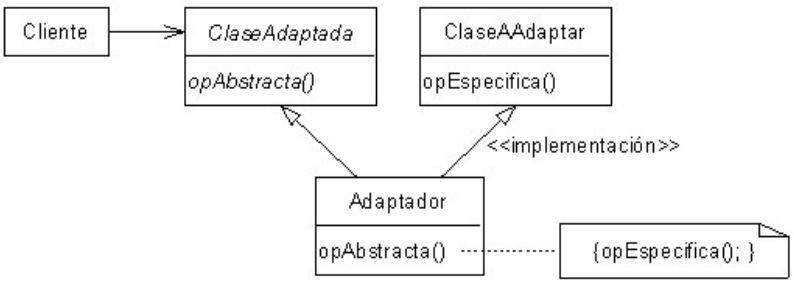


Figura 3.20: Estructura del Adaptador de Clases

La Figura 3.21 ilustra la estructura general del adaptador de objetos. Nótese que en este caso se depende de la composición de objetos.

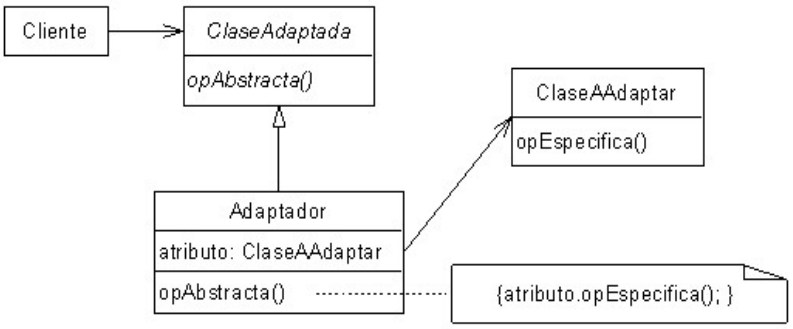


Figura 3.21: Estructura del Adaptador de Objetos

- **Consecuencias de un Adaptador de Clase:**
  - Obliga a una clase a cumplir con una determinada interfaz. No funciona si lo que se quiere es adaptar una clase $y$ sus subclases.
  - Permite modificar parte del comportamiento de la clase a adaptar.
  - Introduce sólo un objeto nuevo.

- **Consecuencias de un Adaptador de Objeto:**

  - Permite a un único adaptador modificar la interfaz de varios objetos adaptados. El adaptador puede además agregar funcionalidad a todos los objetos adaptados.
  - Dificulta la modificación del comportamiento del objeto adaptado.

- **Aplicaciones:**

  Se usan adaptadores en ciertas aplicaciones gráficas, donde coexisten elementos de la interfaz de usuario (*Interactor*: barras de desplazamiento, botones, menús) con objetos gráficos estructurados (*Graphic*: líneas, círculos, polígonos, etc.). Estos dos tipos de elementos tienen apariencia gráfica, pero tienen diferentes interfaces e implementaciones (no comparten clases padres comunes) y por lo tanto son incompatibles. Para solucionar este problema se define un objeto adaptador (*GraphicBlock*), una subclase de *Interactor* que contiene una instancia de *Graphic*. El *GraphicBlock* adapta la interfaz de la clase *Graphic* a la de *Interactor*. El *GraphicBlock* permite que una instancia de *Graphic* sea desplegada, desplazada, y trabajada con un zoom dentro de la estructura de un *Interactor*.

  El Smalltalk estándar define una clase *ValueModel* para vistas que muestran un valor único. *ValueModel* define una interfaz *value, value:* para acceder al valor. Estos son métodos abstractos. Quienes escriben las aplicaciones acceden al valor con nombres más específicos del dominio, como *width* y *width*:, pero no deberían tener que definir subclases de *ValueModel* para adaptar tales nombres específicos del dominio a la interfaz de *ValueModel*. Por el contrario, Smalltalk incluye una subclase de *ValueModel* llamada *PluggableAdaptor*. Un objeto *PluggableAdaptor* adapta otros objetos a la interfaz de *ValueModel* (*value, value:*).

  Otro ejemplo de Smalltalk es la clase *TableAdaptor*. Un *TableAdaptor* puede adaptar una secuencia de objetos a una presentación tabular. La tabla muestra un objeto por fila. El cliente puede parametrizar el *TableAdaptor* con el conjunto de mensajes que usa una tabla para obtener los valores de la columna desde un objeto.

***Otros trabajos de Formalización de Patrones***

Además del mencionado trabajo de Gamma [G+95], existen otras publicaciones dedicadas a la formalización y el análisis de patrones de diseño. Entre ellos se puede mencionar:

- *Software Patterns* (James Coplien, 1996) [Cop96]: define la estrategia basada en patrones, muestra su aplicación en desarrollo de software, presenta ejemplos de implementaciones, y explora las fortalezas y debilidades del enfoque.
- *Design Patterns for Object-Oriented Software Development* (W. Pree y H. Sikora, 1997) [PS97]: dan una visión general del estado del arte en enfoques de diseño basado en patrones, incluyen catálogos de patrones, discuten en detalle unos cuantos patrones útiles, e ilustran la aplicación de patrones de diseño mediante ejemplos.
- *Design Patterns Explained: A New Perspective on Object-Oriented Design* (A. Shalloway y J. Trott, 2004) [ST04]: describen los principios básicos de la programación orientada a objetos y de los patrones de diseño; comienzan con material introductorio sobre desarrollo de software orientado a objetos y las limitaciones del diseño tradicional, para luego explorar diez patrones de diseño usados comúnmente.
- *Design Patterns (Wordware Applications Library)* (Christopher Lasater, 2006) [Las06]: busca mostrar a los diseñadores de software nuevas técnicas y habilidades para mejorar sus trabajos mediante la aplicación de patrones de diseño.

### 3.1.4. Programación Orientada a Objetos

Aunque todas las áreas de tecnologías de objetos han recibido una atención significativa dentro de la comunidad del software, ningún tema ha producido más libros, más discusiones, y más debates que la *Programación Orientada a Objetos* (POO). El punto de vista de la Ingeniería de Software persigue como fin al AOO y el DOO, y considera a la POO (codificación) como actividad importante pero secundaria que es dependiente del análisis y el diseño. La razón para esto es simple. Al incrementarse la complejidad de los sistemas, el diseño de la arquitectura del producto final tiene

una influencia significativamente más fuerte sobre su éxito que el lenguaje de programación usado. A continuación se mencionan algunos lenguajes de programación orientada a objetos, junto con sus caracaterísticas más importantes [Hos00].

### Simula

Fue desarrollado a principios de la década de 1960, como un subconjunto de Algol 60 orientado a la simulación. Sus creadores fueron Kristen Nygaard y Ole-Johan Dahl, del Centro Noruego de Computación en Oslo. Comenzó siendo un lenguaje de programación basado en actividades/procesos, en el cual se declaran diferentes tipos (y comportamientos) de actividades, y luego se pueden crear múltiples procesos para ejecutar las diferentes actividades.

Este diseño original tenía la característica de que, además de tener listas de acciones a ejecutar, los procesos también eran estructuras de datos, y las actividades tenían métodos asociados. Estas actividades y procesos tenían mucha utilidad aparte de la pura simulación, y cuando se distribuyó Simula-67, éstas habían sido renombradas como "clases" y "objetos" (ahí nació la programación "orientada a objetos"). Simula-67 fue lanzado oficialmente por sus autores en mayo de 1967, en la Conferencia de Trabajo en Lenguajes de Simulación [DMN67], en Lysebu, cerca de Oslo.

Además de las características de Algol 60, Simula agregó soporte para Objetos (como clausuras que retornan referencias a sí mismos) con estado protegido, herencia simple para subtipado y compartición de código, clases parcialmente abstractas, sobreescritura de métodos, y clausuras anidadas (incluyendo procedimientos y clases anidados, y clases locales a procedimientos). Pero Simula-67 no soporta transferencia dinámica, algo que la mayoría considera necesario para la programación orientada a objetos "verdadera". Un objeto debe ser chequeado en tiempo de ejecución, y luego se puede acceder al atributo o método apropiado.

### Smalltalk

También tuvo sus orígenes en la década de 1960, pero tuvo varias redefiniciones antes de ser definitivamente lanzado al público en 1980. Fue desarrollado por un grupo liderado por Alan Kay en el Centro de Investigación de Xerox en Palo Alto. En [GR83b] se presenta el lenguaje Smalltalk-80 y su implementación, y ya en el año 1996 [Kay96] su creador narra los primeros años de desarrollo del lenguaje.

Es un lenguaje sin tipos, basado en clases. Smalltalk distingue entre los atributos y métodos que pertenecen a una clase y los que pertenecen a las instancias. Los métodos de *clases* se localizan en el meta-objeto de la clase y sólo se pueden referir a atributos de la clase (también localizados en el meta-objeto para proveer un estado compartido por todas las instancias). Los métodos de *instancia* se mantienen en forma local a cada objeto y puede hacer referencia tanto a las variables de clase como a las variables de instancia (que proveen el estado local). Todos los métodos son públicos, mientras que todos los atributos son privados.

Se provee herencia simple, junto con clases (parcialmente) abstractas, y sobrescritura de métodos (incluyendo la modificación de la signatura). Aunque Smalltalk es no tipado, el propósito principal de la herencia no es la simple compartición de código. El principio subyacente es similar al subtipado para proveer especialización de objetos, donde se puede usar una instancia de una subclase como si fuera una instancia de la superclase.

### Modula-3

Fue diseñado por Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow y Greg Nelson en el Centro de Investigación de Sistemas de DEC (DECSRC) y en Olivetti a finales de los 80. La definición inicial del lenguaje fue publicada en agosto de 1988, fue revisada en enero de 1989 [C$^+$89] en base a las recomendaciones de los implementadores, y luego tuvo otra revisión en el año 1992 [C$^+$92].

Modula-3 es un lenguaje basado en clases, en el que los nombres de clase actúan como nombres de tipos – no hay definiciones o declaraciones de tipos explícitas. Es fuertemente tipado, sin conversión automática ni inferencia de tipos. Además, la igualdad de tipos se define en base a la estructura de tipos/clases, y no a su nombre.

Las definiciones de clase son "parcialmente opacas", es decir que los métodos y atributos pueden o no ser visibles a otras clases. Se provee herencia simple como un medio de especialización de tipos (permitiendo que las subclases sobrescriban los métodos de su superclase) que también posibilita la reutilización de código, y es posible usar clases abstractas como un medio de especificar tipos (que contienen sólo declaraciones y signaturas de métodos) sin implementaciones. Modula-3 también provee hilos de peso ligero, excepciones, módulos, e interfaces de módulos.

### Self

La primera implementación pública de Self fue distribuida en 1991, aunque el lenguaje fue inicialmente diseñado en 1986 por David Ungar y Randall Smith y presentado en 1987 [US87] en la Conferencia sobre Sistemas, Lenguajes y Aplicaciones de Programación Orientada a Objetos (OOPSLA), en Orlando, Florida, EE.UU. Su mayor influencia es la de Smalltalk, además de varios lenguajes de investigación basados en prototipos.

Self es un lenguaje sin clases, que usa objetos prototipo, y clonación para construir nuevos objetos. El ambiente de ejecución es responsable de realizar un chequeo dinámico de tipos, y no hay tipos estáticos o declaraciones de tipos requeridas (ni incluidas) en el lenguaje.

El principio fundamental de Self es *"Mensajes al Final"*. Todas las operaciones se implementan como mensajes. Cada objeto se compone enteramente de *slots*, que contienen estado o comportamiento. Cuando un objeto recibe un mensaje, se chequean los *slots* del mismo para el correspondiente mensaje. Si el mensaje no se encuentra en uno de los *slots* del objeto, entonces se accede al puntero en el *slot* denominado "padre" (que cada objeto tiene como resultado de la clonación), y la búsqueda del mensaje recomienza en los punteros del padre. Una vez que se encuentra el *slot* apropiado, el contenido del mismo puede ser un puntero a otro objeto (una variable) o un método.

Self provee características como herencia, estado compartido, compartición de código y transferencia dinámica. Además, cada *slot* de un objeto puede verse como una dirección de memoria que puede reasignarse a voluntad. Así, no sólo se pueden modificar métodos en el medio de la ejecución de un programa, sino que también se pueden cambiar atributos por métodos y viceversa.

### Eiffel

Eiffel fue diseñado por Bertrand Meyer, quien concibió las primeras ideas en 1985. La distribución al público llegó en octubre de 1986, como ISE Eiffel 1, y ha evolucionado sin interrupciones hasta hoy. En el año 1991 Meyer publica una descripción formal del lenguaje [Mey91]. Es un lenguaje basado en clases, en el cual las definiciones de Tipo y Clase son idénticas. La equivalencia de tipos se basa en la equivalencia de nombres de clases.

Las clases pueden contener (múltiples) estipulaciones de prestaciones, que a su vez pueden contener múltiples Atributos/valores y Rutinas/procedimientos. La clasificación de una prestación dada (Rutina o Atributo) es desconocida para las otras clases, es decir, un Atributo de tipo T tiene la misma "apariencia" que una Rutina que no tiene argumentos y que devuelve un elemento de tipo T. Cada lista de prestaciones tiene una lista de clientes asociada, que especifica las clases a las que se les permite acceder a las mismas. Las clases también pueden diferir la implementación de cualquier prestación, haciéndola una clase abstracta.

Eiffel soporta herencia múltiple (incluyendo reuso de código), donde el compilador fuerza el cambio de nombre de las prestaciones en caso de colisión. Además, Eiffel permite al programador no sólo redefinir (o indefinir) la implementación de prestaciones particulares, sino también modificar la lista de clientes de las prestaciones heredadas.

Una de las características más preponderantes de Eiffel es el soporte integrado para *"Diseño por Contrato"*, que no es necesariamente orientado a objetos, pero que funciona bien en procesos de diseño orientados a objetos. Eiffel provee *Aserciones* en la forma de Invariantes sobre objetos, y Pre y Poscondiciones sobre rutinas individuales. En general, una aserción es una expresión lógica que no tiene efecto si es verdadera, pero de otro modo resulta en una Excepción en tiempo de ejecución.

Las aserciones son extremadamente potentes cuando se combinan con la herencia. Eiffel requiere no sólo que los invariantes de todas las superclases sean compatibles, sino también que la redefinición de cualquier característica adhiera a las pre y poscondiciones iniciales de la superclase, o que tenga una precondición más débil y una poscondición más fuerte.

### Sather

Inicialmente distribuido en junio de 1991 por el ICSI (Instituto Internacional de Ciencias de la Computación) de la Universidad de California en Berkeley, Sather comenzó a gestarse a mediados de 1990, y sufrió influencias de una amplia variedad de lenguajes, aunque el principal fue Eiffel. Su desarrollo estuvo a cargo de un grupo internacional liderado por Stephen Omohundro, que publicó la especificación de la primera versión del lenguaje junto a David Stoutamire en 1996 [SO96].

Sather es un lenguaje basado en clases que provee atributos y rutinas públicos y privados (además de atributos de sólo lectura). Pero a diferencia de Eiffel, no permi-

te listas de acceso explícitas para las prestaciones. También provee soporte para el principio de "Diseño por Contrato", incluyendo pre y poscondiciones, invariantes, e imposiciones generalizadas, que son declaraciones que pueden aparecer en cualquier bloque de código y resultar en un error fatal si no se evalúan como verdaderas.

La herencia se divide en dos nociones: subtipado e inclusión de código. Se provee el *subtipado* netamente como un método de especialización de tipos. Una clase concreta puede ser declarada como un subtipo de una clase abstracta, y todos los objetos de la clase conforman automáticamente con el tipo de la clase abstracta. Cada clase concreta puede ser subtipo de una clase abstracta como máximo, pero las clases abstractas pueden ser subtipos de cualquier número de otras clases abstractas. La *inclusión de código* permite que una clase importe directamente la implementación de otras (múltiples) clases, con el propósito de reusar el código (sin ningún efecto sobre el tipo de la clase que incluye el código). La clase puede redefinir, indefinir, renombrar, o modificar los permisos de acceso de cualquier rutina o atributo incluido, y el compilador fuerza el cambio de nombre cuando hay conflictos. Aquí también existen clases parciales, que no tienen tipo (y por lo tanto no pueden ser instanciadas) pero pueden ser incluidas en cualquier número de clases concretas.

La característica más notable de Sather es la inclusión del tipo "Same", que es el "Auto Tipo" raramente implementado en los lenguajes actuales.

### C++

"C con clases" fue distribuido en 1980 como una versión mejorada de C que incluía clases para abstracción de datos. La primera versión de C++ fue lanzada en 1983, como un derivado de C "más verdaderamente" orientado a objetos. Bjarne Stroustrup, el diseñador e implementador original de C++, publicó la especificación del lenguaje en el año 1986 [Str86].

C++ es un lenguaje basado en clases, diseñado para permitir al programador un control de muy bajo nivel sobre la estructura y el acceso de los objetos. Las características orientadas a objetos de C++ incluyen: funciones virtuales (abstractas), que resultan en clases virtuales, control de acceso público/privado/protegido sobre funciones y atributos miembros individuales, clases amigas (para permitir que clases explícitamente designadas accedan a estados privados), clases anidadas, herencia múltiple (para subtipado y compartición de código) con redefinición de métodos, y

clases y funciones plantilla (genéricas).

Otras características generales incluyen: memoria controlada por el usuario (el heap), referencias directas a memoria, chequeo estático de tipos, sobrecarga de métodos, excepciones, hilos, y espacios de nombres explícitamente construidos. Desafortunadamente, la especificación del lenguaje no profundiza en la explicación de los detalles de cómo interactúan estas características, y muchas implementaciones son incompatibles.

### Java

Este lenguaje surgió para programar software de sistemas embebidos para dispositivos electrónicos inteligentes, en Sun Microsystems. Alrededor de esa época (año 1993), el uso de la WWW estaba en franco crecimiento, y Sun descubrió múltiples usos para su código pequeño, seguro e independiente de la plataforma. Su creador, James Gosling, publicó en 1996 la especificación del lenguaje [GJS96].

Java es un lenguaje basado en clases, y las ideas de velocidad, independencia de la plataforma y seguridad en tiempo de ejecución son cruciales en su diseño. Cuando la motivación detrás del lenguaje pasó a ser la WWW, la idea de programas distribuidos adquirió extrema importancia. Una de las principales características que provee Java es el sistema de Invocación Remota de Métodos (RMI), que permite la invocación de métodos y el intercambio de objetos entre máquinas virtuales (incluso a través de la red) de manera semi transparente.

Además del RMI, Java soporta características generales de programación tales como excepciones, recolección de basura (que se considera crucial), verificación del código de bytes (que valida la seguridad de un programa determinado), hilos, sobrecarga de métodos, y paquetes (para crear espacios de nombres). Como lenguaje orientado a objetos, Java soporta múltiples niveles de ocultamiento de la implementación, clases parcialmente abstractas, clases finales (de las que no se puede heredar) y variables (clases) estáticas.

Se provee herencia simple de clases para subtipado y compartición de código, además de herencia múltiple de interfaces, que actúan como declaraciones de tipo, o clases completamente abstractas. También soporta clases anidadas y anónimas.

### 3.1.5. Pruebas Orientadas a Objetos

El objetivo de la realización de pruebas en la Ingeniería de Software es, indicado de manera simple, encontrar el mayor número posible de errores con una cantidad de esfuerzo racional a lo largo de un espacio de tiempo realista. Aunque este objetivo fundamental permanece igual para el software orientado a objetos, la naturaleza de los programas OO cambia la estrategia y las tácticas de la prueba.

Pudiera argumentarse que, con la madurez del AOO y el DOO, el mayor grado de reusabilidad (reutilización) de patrones de diseño reduciría la necesidad de pruebas pesadas en sistemas OO. Pero en la realidad se da exactamente lo opuesto. Binder [Bin94a] examina esto cuando comenta:

> Cada reutilización es un nuevo contexto de uso y es prudente repetir las pruebas. Parece en consecuencia, que no se requerirán menos pruebas para obtener una alta confiabilidad en sistemas orientados a objetos.

Para probar adecuadamente los sistemas OO, deben hacerse tres cosas: (1) la definición de las pruebas debe ampliarse para incluir técnicas de detección de errores aplicables a los modelos de DOO y AOO; (2) la estrategia para las pruebas de unidad e integración deben cambiar significativamente; (3) el diseño de casos de prueba debe tener en cuenta las características propias del software orientado a objetos.

#### Ampliando la visión de la realización de la prueba

La revisión de los modelos de análisis y diseño OO es especialmente útil, pues las mismas construcciones semánticas (por ejemplo: clases, atributos, operaciones, mensajes) aparecen en los niveles de análisis, diseño y codificación. Por esto, un problema en la definición de atributos de clases no descubierto durante el análisis provocará la propagación de efectos colaterales, si el problema no se descubriera hasta el diseño o codificación (o incluso hasta la próxima iteración del análisis).

Durante las etapas avanzadas de su desarrollo, los modelos de AOO y DOO proporcionan información sustancial acerca de la estructura y comportamiento del sistema. Por esta razón, estos modelos deberán ser sometidos a una revisión rigurosa, previa a la generación del código. Todos los modelos orientados a objetos deberán demostrar su correctitud, completitud y consistencia [MK94]. Con esta finalidad pueden usarse las revisiones técnicas formales, ya que al no poder ejecutarse, estos modelos no

pueden probarse en el sentido convencional.

La *correctitud sintáctica* se refiere al uso apropiado de la simbología y de las convenciones de modelado propias del método específico de análisis y diseño elegidos para el proyecto. La *correctitud semántica* se traduce en la conformidad del modelo con el dominio del problema del mundo real. El modelo está semánticamente correcto si refleja el mundo real de manera exacta.

La *consistencia* de los modelos de AOO y DOO pueden juzgarse a través de una "consideración de las relaciones entre las entidades del modelo. Un modelo inconsistente tiene representaciones por una parte que no son correctamente reflejadas en otras partes del modelo" [MK94]. Para valorar la consistencia, deberán examinarse cada clase y sus conexiones a otras clases. También deberán revisarse los subsistemas que componen el producto, la manera en la cual los subsistemas se asignan a los procesadores, y la asignación de clases a subsistemas. Y en el modelo de objetos, se debe evaluar la actividad de intercambio de mensajes necesarios para implementar las colaboraciones entre clases.

***Diseño por Contrato y Aserciones***

La correctitud de un programa se define como la coincidencia entre su comportamiento deseado (especificación) y su comportamiento real. Pero para poder hablar de correctitud, es necesario describir con precisión la relación existente entre los datos proporcionados y los resultados esperados.

El comportamiento esperado de un algoritmo puede expresarse mediante lo que se denomina *especificación pre-post* de programas. La idea es que, si se supone que al comienzo los datos cumplen con la precondición, entonces al final los resultados cumplen con la postcondición.

La *precondición* describe las condiciones iniciales que deben cumplir los datos de entrada. Expresa las restricciones bajo las cuales la operación funcionará correctamente. Un sistema correcto nunca realizará una operación en un estado que no cumpla con la precondición de dicha operación. La *postcondición* describe la relación entre los datos de entrada y los resultados obtenidos. Expresa cuál es el estado resultante de realizar la operación, supuesto que en su inicio se cumplía la precondición.

Entonces, una especificación pre-post define un *contrato* entre una operación (*proveedor*) y quien la invoca (*cliente*). Si el cliente se compromete a llamar a la operación cumpliendo la precondición, la operación se compromete a devolverle un estado que

cumple la postcondición. Por lo tanto, la precondición constituye una obligación del cliente y un beneficio para el proveedor, mientras que la postcondición es una obligación del proveedor y un beneficio para el cliente.
Ejemplos:

- Operación: $x := x + 5$
  $\text{Pre} = \{x \geq 9\}$ $\quad$ $\text{Pos} = \{x \geq 14\}$

- Operación: Sumar los elementos de un arreglo $A(1..n)$ de enteros.
  $\text{Pre} = \{n \geq 1\}$ $\quad$ $\text{Pos} = \{s = \sum_{i=1}^{n} A(i)\}$

Las *aserciones* son expresiones lógicas (o fórmulas) incorporadas al texto de un programa como comentarios. Estas expresiones están asociadas a puntos del programa, y expresan propiedades de las variables del mismo. La inclusión de aserciones ayuda a aclarar las especificaciones de un programa, y disminuye el tiempo insumido en la depuración y en la localización de errores.

Supongamos por ejemplo que un método $M$ calcula el cociente $q$ y el resto $r$, como resultado de una división entera entre $x$ e $y$. Las aserciones para este método, en forma de pre y postcondiciones, serían las siguientes:

$$\text{Pre} = \{y > 0 \wedge x \geq 0\} \qquad M \qquad \text{Pos} = \{x = q * y + r \wedge y > r \geq 0 \wedge q \geq 0\}$$

Esto significa que si $M$ comienza en un estado que satisface las precondiciones, entonces termina en un estado que satisface las postcondiciones. Las precondiciones son las condiciones *mínimas* (más débiles) que deben cumplir los datos, mientras que las postcondiciones son *relaciones* (más fuertes) entre los datos y los resultados.

### Estrategias de Pruebas Orientadas a Objetos

La estrategia clásica para ejecutar pruebas sobre software de computadora comienza con la "prueba a pequeña escala" y se dirige hacia la "prueba a gran escala". Dicho en la jerga de prueba del software, se comienza con la *prueba de unidad*, y se culmina con la *validación y prueba del sistema*.

#### *Prueba de Unidad en el Contexto OO*

Al considerar el software orientado a objetos, cambia el concepto de unidad. En vez de módulos individuales, la menor unidad a probar es la clase u objeto encapsulado,

que empaqueta los atributos y las operaciones que los manipulan. El significado de prueba de unidad, por tanto, cambia dramáticamente. Deja de referirse a la prueba en detalle de una operación aislada (la vista convencional de prueba de unidad), para apuntar a la *prueba de clases*, dirigida por las operaciones encapsuladas por la clase y el estado del comportamiento de la misma.

***Prueba de Integración en el Contexto OO***

Debido a que el software orientado a objetos no tiene una estructura de control jerárquica, las estrategias convencionales de integración ascendente y descendente poseen escaso significado. Existen dos estrategias diferentes para pruebas de integración en sistemas OO [Bin94b]. La primera, *pruebas basadas en hilos*, integra el conjunto de clases necesario para responder a una entrada o evento del sistema. Cada hilo se integra y prueba individualmente. Se aplica la prueba de regresión para asegurar que no ocurren efectos colaterales. El segundo enfoque para la integración, *pruebas basadas en uso*, comienza la construcción del sistema probando aquellas clases (llamadas *clases independientes*) que usan muy pocas (si alguna) de las clases servidor. Después de probar las clases independientes, se comprueba la próxima capa de clases, llamadas *clases dependientes*, que hacen uso de las clases independientes. Esta secuencia continúa hasta construir el sistema por completo.

***Prueba de Validación en un Contexto OO***

En el nivel de validación del sistema, los detalles de conexiones de clases desaparecen. Como en el caso del software convencional, la validación del software orientado a objetos se centra en las acciones visibles del usuario y las salidas del sistema reconocibles por éste. Para asistir en la determinación de pruebas de validación, el ejecutor de la prueba debe basarse en los casos de uso que forman parte del modelo de análisis. El caso de uso brinda un escenario que posee una alta probabilidad de errores encubiertos en los requisitos de interacción del cliente. Adicionalmente, se pueden derivar casos de prueba a partir del modelo objeto-comportamiento y del diagrama de flujo de eventos como parte del AOO.

### Diseño de Casos de Prueba para Software OO

A diferencia del diseño de casos de prueba convencionales, que se orienta a través de una visión entrada–procesamiento–salida del software o por el detalle algorítmico

de los módulos individuales, la prueba orientada a objetos se centra en el diseño de secuencias de operaciones apropiadas para ejercitar el estado de una clase.

Por lo tanto, es necesario tener presente que las características propias de la metodología de objetos tienen importantes derivaciones en el momento de diseñar los casos de prueba. Por ejemplo, el encapsulamiento puede dificultar la obtención de información del estado interno de un objeto. La herencia fuerza la repetición de las pruebas en cada nuevo contexto de uso, sin mencionar las complicaciones adicionales que puede acarrear la herencia múltiple.

Berard [Ber93] ha sugerido un enfoque general para el diseño OO de casos de prueba:

1. Cada caso de prueba debe estar identificado unívocamente y asociado explícitamente con el caso de uso a probar.

2. Debe establecerse el propósito de la prueba.

3. Se debe desarrollar una lista de pasos de prueba para cada prueba, la cual deberá contener:

   *a*) Una lista de estados especificados para los objetos a probar.

   *b*) Una lista de mensajes y operaciones a ejercitar como consecuencia de la prueba.

   *c*) Una lista de excepciones que pueden ocurrir al probar el caso.

   *d*) Una lista de condiciones externas (por ejemplo: cambios en el entorno externo al software que debe existir para conducir propiamente la ejecución de la prueba).

   *e*) Información suplementaria que ayudará en la comprensión o implementación de la prueba.

### *Diseño de Pruebas Basadas en Escenarios*

Las *pruebas basadas en escenarios* se concentran en lo que hace el usuario, no en lo que hace el producto. Esto significa capturar las tareas (a través de casos de uso) que el usuario debe realizar, y después aplicarlas a sus variantes como pruebas. Este tipo de pruebas intenta verificar que el producto haga lo que desea el cliente, y que no haya errores asociados con la interacción entre subsistemas.

Estos casos de prueba deben ser lo bastante complejos y realistas como para capturar las especificaciones incorrectas (ya que la calidad del producto, es decir, su conformidad con los requerimientos, se ve disminuida) y los errores asociados con la interacción entre subsistemas (que ocurren cuando el comportamiento de un subsistema crea circunstancias, como eventos o flujos de datos, que provocan que otro subsistema falle). Las pruebas basadas en escenarios tienden a utilizar varios subsistemas en una prueba simple (los usuarios no se limitan al uso de un sólo subsistema en un momento dado).

---

**Ejemplo: Caso de Prueba**

Los ingenieros de pruebas sugieren un conjunto de casos de prueba para el caso de uso *Pagar Factura*, en el que cada caso de prueba verificará un escenario del caso de uso. Uno de los casos de prueba propuestos es el pago de una factura por una orden de $300 de una bicicleta. A este caso de prueba le llaman *Pagar 300–Bicicleta*.

Para estar completo, este caso de prueba tiene que especificar las entradas, el resultado esperado, y las demás condiciones relevantes.

**Entradas**

- Existe una orden válida de una bicicleta, que ha sido propuesta al vendedor, *Bicicletas de Montaña S. A.* El precio de lista de la bicicleta es de $300, incluyendo el envío.
- El comprador ha recibido una confirmación de orden (Nº 98765) de una bicicleta. El precio confirmado es de $300, incluyendo el envío.
- El comprador también ha recibido una factura (Nº 12345). La factura obedece a la confirmación de orden de la bicicleta. Esta es la única factura presente en el sistema. El monto facturado debería ser por un total de $300, y la factura debería estar en estado *Pendiente*. La factura debería apuntar a una cuenta bancaria 22-222-2222, que debería recibir el dinero. La cuenta tiene un saldo actual de $963.456,00. La cuenta debería pertenecer al vendedor.
- La cuenta del comprador 11-111-1111 muestra un saldo de $350.

**Resultado**

- El estado de la factura debería pasar a *Cerrada* (para indicar que fue pagada).

- La cuenta del comprador 11-111-1111 debería mostrar un saldo de $50.
- El saldo de la cuenta del vendedor 22-222-2222 debería haber aumentado para ser ahora de $963.756,00.

**Condiciones**

- No se le permite a ningún otro caso de uso (instancia) acceder a las cuentas durante este caso de prueba.

---

Cada caso de prueba tiene asociado uno o varios *procedimientos de prueba*. Un procedimiento de prueba especifica cómo ejecutar uno o varios casos de prueba. Es decir que con frecuencia es útil reusar un procedimiento de prueba para varios casos de prueba, y reusar varios procedimientos de prueba para un mismo caso de prueba.

---

**Ejemplo: Procedimiento de Prueba**

Se requiere un procedimiento de prueba para que un individuo ejecute el caso de prueba *Pagar 300–Bicicleta* del ejemplo anterior. La primer parte del procedimiento de prueba se especifica como sigue (no es necesario incluir el texto entre corchetes en la especificación, porque ya está especificado en el caso de prueba):

Caso de prueba soportado: *Pagar 300–Bicicleta*.

1. De la ventana principal, seleccionar el menú *Explorar Facturas*. Se abre la ventana de diálogo *Consulta de Facturas*.
2. En el campo *Estado de Factura*, seleccionar *Pendiente* y pulsar el botón *Consultar*. Se despliega la ventana de *Resultados de la Consulta*. Verificar que la factura especificada en el caso de prueba [Nº 12345] aparezca en la ventana de *Resultados de la Consulta*.
3. Elegir la factura especificada a pagar, haciéndole doble-clic. Se despliega la ventana de *Detalles de la Factura* para la factura elegida. Verificar los siguientes campos:
    - Que *Estado* sea *Pendiente*.
    - Que *Fecha de Pago* esté vacío.

- Que *Nº de Confirmación de Orden* coincida con el Nº especificado en el caso de prueba [Nº 98765].
- Que *Monto de la Factura* coincida con el monto especificado en el caso de prueba [$300].
- Que *Cuenta* coincida con la cuenta especificada en el caso de prueba [22-222-2222].

4. Seleccionar el casillero de verificación *Autorizar el Pago* para iniciar el pago de esta factura. Se despliega la caja de diálogo *Pago de Factura*.

5. Etc. (se especifica cómo se ejecuta el procedimiento completo del caso de uso *Pagar Factura* mediante la interfaz de usuario, dándole ciertas entradas al sistema, y qué se debe verificar en la salida del sistema).

---

***Pruebas Basadas en Fallos***

El objetivo de la prueba basada en fallos dentro de sistemas OO es el de diseñar pruebas que posean una alta probabilidad en la detección de posibles errores. Se deben probar los casos extremos, ya que es en ellos donde los ingenieros del software comenten fallos más a menudo.

Las pruebas de integración buscan errores posibles en la comunicación de mensajes. Se han encontrado tres tipos de errores en este contexto: resultado inesperado, operación/mensaje incorrecto, invocación incorrecta. Para determinar errores posibles al invocar funciones (operaciones), debe examinarse el comportamiento de la operación.

Las técnicas de integración son aplicables tanto a atributos como a operaciones. Los "comportamientos" de un objeto se definen por los valores asignados a sus atributos. La prueba debe ejercitar a los atributos para determinar si existen valores apropiados para los distintos comportamientos del objeto.

Es importante notar que las pruebas de integración tratan de encontrar errores en el objeto cliente, no en el servidor. El objetivo de la prueba de integración es determinar si existen errores en el código que invoca, no en el invocado.

### Métodos de Prueba Aplicables al Nivel de Clase

Señalamos que las pruebas del software comienzan "a lo pequeño" y lentamente progresan hacia la ejecución de pruebas "de más nivel". La realización de pruebas desde lo más elemental para sistemas OO se centran en una clase simple y los métodos encapsulados en ella. Las *pruebas aleatorias* y el *particionamiento* son métodos que pueden usarse para ejercitar una clase durante las pruebas OO [KT94].

#### *Pruebas Aleatorias para Clases OO*

Para ilustrar brevemente estos métodos, se considera una aplicación bancaria en la cual una clase *cuenta* posee las siguientes operaciones: *abrir, preparar, depositar, retirar, obtenerBalance, resumir, limitarCrédito*, y *cerrar*. Cada una de estas operaciones puede aplicarse para *cuenta*, pero con ciertas restricciones derivadas de la naturaleza del problema (por ejemplo, la cuenta debe estar abierta antes que puedan aplicarse otras operaciones y cerrada después que se han completado todas las operaciones). Incluso con estas restricciones, existen muchas permutaciones de las operaciones. El mínimo comportamiento de la historia de vida de una instancia de *cuenta* incluye las siguientes operaciones:

*Abrir-preparar-depositar-retirar-cerrar*

Esto representa la *secuencia mínima de prueba* para *cuenta*. Sin embargo, dentro de la siguiente secuencia, puede darse una amplia variedad de comportamientos:

*Abrir - preparar - depositar* [*depositar* | *retirar* | *obtenerBalance* | *resumir* | *limitarCredito*]$^n$ - *retirar - cerrar*

Puede generarse aleatoriamente una amplia variedad de secuencias de operaciones. Se ejecutarán estas y otras pruebas de orden aleatorio para ejercitar las historias de vida de diferentes instancias de clase.

#### *Pruebas de Partición al Nivel de Clase*

La *partición basada en estados* categoriza las operaciones de clase basándose en su capacidad para cambiar el estado de la misma. Si se considera de nuevo la clase *cuenta*, las operaciones de estado incluyen *depositar* y *retirar*, mientras que las operaciones *obtenerBalance*, *resumir* y *limitarCrédito* no son de estado. Las pruebas se diseñan de manera tal que se ejerciten por separado las operaciones que cambian de estado y las que no lo hacen.

La *partición basada en atributos* divide las operaciones de clase basándose en

los atributos que usan. Para la clase *cuenta*, los atributos *balance* y *límiteCrédito* pueden utilizarse para definir particiones: (1) operaciones que usan el *límiteCrédito*, (2) operaciones que modifican el *límiteCrédito*, y operaciones que no usan ni modifican el *límiteCrédito*. Se diseñan entonces secuencias de pruebas para cada partición.

La *partición basada en categorías* separa las operaciones de clases basándose en las funciones genéricas que cada una realiza. Por ejemplo, las operaciones en la clase *cuenta* pueden categorizarse en operaciones de inicialización (*abrir, preparar*), operaciones computacionales (*depositar, retirar*), solicitudes (*obtenerBalance, resumir, limitarCrédito*) y operaciones de terminación (*cerrar*).

### Diseño de Casos de Prueba Interclases

El diseño de casos de prueba se torna más complicado cuando comienza la integración del sistema OO. Es en esta etapa cuando deben comenzar las pruebas de las colaboraciones entre clases. Como en la prueba de clases individuales, las pruebas de colaboraciones entre clases pueden realizarse a través de la aplicación de métodos aleatorios y de particionamiento, así como a través de pruebas basadas en escenarios y pruebas de comportamiento.

#### *Pruebas de Clases Múltiples*

Kirani y Tsai [KT94] sugieren la siguiente secuencia de pasos para generar casos de prueba aleatorios para múltiples clases:

- Para cada clase cliente, usar la lista de operadores de clase para generar una serie de secuencias de prueba aleatorias. Los operadores enviarán mensajes a otras clases servidoras.
- Para cada mensaje que se genera, determinar la clase colaboradora y el operador correspondiente en el objeto servidor.
- Para cada operador en el objeto servidor (que haya sido invocado por los mensajes enviados desde el objeto cliente), determinar los mensajes que éste transmite.
- Para cada uno de los mensajes, determinar el próximo nivel de operadores que se invocan e incorporarlos en la secuencia de prueba.

El enfoque para pruebas de partición de clases múltiples es similar al usado para la prueba de partición de clases individuales. Una clase simple se divide como se examinó en la Sección 3.1.5. Sin embargo, la secuencia de prueba se expande para incluir aquellas operaciones que se invocan a través de los mensajes a las clases colaboradoras. Un enfoque alternativo divide las pruebas basándose en las interfaces a una clase particular. Para refinar aún más las particiones, se puede usar el particionamiento basado en estados.

***Pruebas Derivadas de Modelos de Comportamiento***

El diagrama de transición de estados (DTE) de una clase es un modelo que representa el comportamiento dinámico de la misma. El DTE para una clase puede usarse para ayudar a derivar una secuencia de pruebas que ejercitan el comportamiento dinámico de la clase (y aquellas clases que colaboran con ella). La Figura 3.22 [KT94] ilustra un DTE para la clase *cuenta* examinada anteriormente. En referencia a la figura, las transiciones iniciales se mueven desde el estado *vaciar cuenta* al de *preparar cuenta*. La mayoría de los comportamientos para instancias de la clase ocurren al encontrarse en el estado *trabajo con la cuenta*. Una acción final como *retirarTodo* o *cerrar* provoca que la clase *cuenta* realice transiciones a los estados *cuenta no operativa* y *eliminar cuenta*, respectivamente.

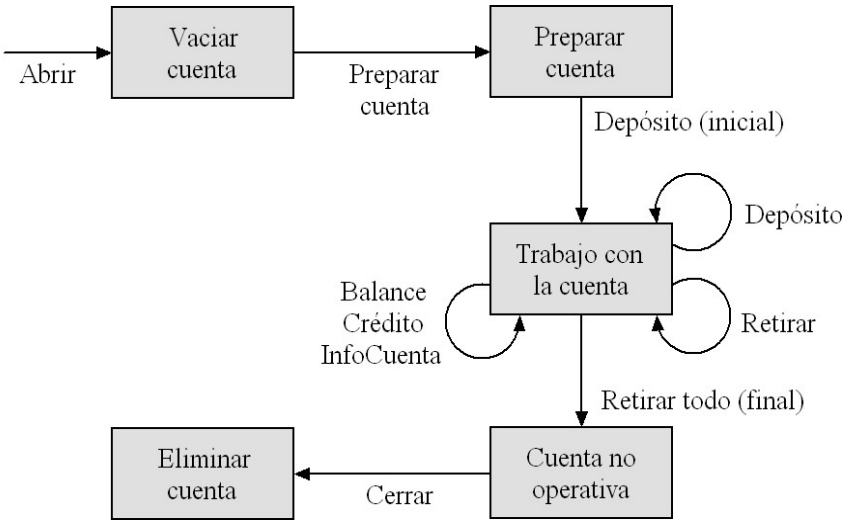


Figura 3.22: Diagrama de transición de estados para la clase *cuenta*

Las pruebas a diseñar deberán ser capaces de *abarcar todos los estados*. Esto es, las secuencias de operaciones deberán provocar que la clase *cuenta* ejecute transiciones a

través de todos los estados permitidos. En situaciones en las cuales el comportamiento de la clase consiste en colaborar con una o más clases, se usan múltiples DTE para registrar el flujo de comportamiento del sistema.

El modelo de estado puede recorrerse de forma *primero a lo ancho* [MK94]. En este contexto, primero a lo ancho implica que un caso de prueba ejercita una transición simple y que cuando se debe probar una nueva transición solamente se usan las transiciones previamente probadas.

## 3.2. El Proceso Unificado de Desarrollo de Software

### 3.2.1. Introducción

A principios de los años 90s existían numerosas metodologías orientadas a objetos. De hecho, como ya se vio en la Sección 3.1.3, durante ese período Grady Booch desarrolló su método [Boo91], y James Rumbaugh fue el principal creador de la Técnica de Modelado de Objetos (OMT) [R$^+$91]. Se hacía evidente la necesidad de un lenguaje visual uniforme y consistente para expresar los resultados de todas las metodologías existentes.

Los dos autores mencionados, junto con Ivar Jacobson, que se unió a ellos en 1995, trabajaron en la unificación de sus métodos y en la creación de un lenguaje de modelado unificado. Como resultado de sus esfuerzos, sobre el final de la década de los 90s publicaron su Proceso Unificado de Desarrollo de Software [JBR99] y su Lenguaje de Modelado Unificado (UML) [BRJ98, RJB98].

Un *proceso* es un conjunto de etapas parcialmente ordenadas con las que se pretende alcanzar un objetivo. En este caso, el objetivo será *transformar en forma eficiente y predecible los requerimientos de un usuario en un producto de software que se ajuste a sus necesidades.*

#### Principios Básicos

- Dirigido por Casos de Uso

  La razón de ser de un sistema de software es servir a usuarios, ya sean humanos u otros sistemas; un *caso de uso* es una facilidad que el software debe proveer a sus usuarios. Los casos de uso reemplazan la antigua especificación

funcional tradicional y constituyen la guía fundamental establecida para las actividades a realizar durante todo el proceso de desarrollo, incluyendo el diseño, la implementación y las pruebas del sistema.

- Centrado en la Arquitectura

  La *arquitectura* involucra los elementos más significativos del sistema y está influenciada entre otros por plataformas de software, sistemas operativos, gestores de bases de datos, protocolos, consideraciones de desarrollo como sistemas heredados y requerimientos no funcionales. Los casos de uso guían el desarrollo de la arquitectura y la arquitectura se realimenta en los casos de uso; los dos juntos permiten conceptualizar, gestionar y desarrollar adecuadamente el software.

- Iterativo e Incremental

  Para hacer más manejable un proyecto se recomienda dividirlo en ciclos. Para cada ciclo se establecen fases de referencia, cada una de las cuales debe ser considerada como un *miniproyecto* cuyo núcleo fundamental está constituido por una o más iteraciones de las actividades principales básicas de cualquier proceso de desarrollo.

**Otras Características**

- Desarrollo Basado en Componentes

  La creación de sistemas de software requiere dividir el sistema en componentes con interfaces bien definidas, que posteriormente serán ensamblados para generar el sistema. Esta característica en un proceso de desarrollo permite que el sistema se vaya creando a medida que se obtienen o que se desarrollan y maduran sus componentes.

- Proceso Integrado

  Se establece una estructura que abarca los ciclos, fases, flujos de trabajo, mitigación de riesgos, control de calidad, gestión del proyecto y control de configuración; el Proceso Unificado establece una estructura que integra todas estas facetas. Además esta estructura cubre a los vendedores y desarrolladores de herramientas para soportar la automatización del proceso, soportar flujos individuales de trabajo, para construir los diferentes modelos e integrar el trabajo a través del ciclo de vida y a través de todos los modelos.

- Utilización de un Lenguaje Unificado de Modelamiento
  Se adopta UML como único lenguaje de modelamiento para el desarrollo de todos los modelos.

### Personas, Proyecto, Producto, y Proceso

#### *Personas*

Son los arquitectos, desarrolladores, ingenieros de prueba, personal de gestión, usuarios, clientes, y demás involucrados en un proyecto de software. Son seres humanos reales. Las personas están implicadas en el desarrollo de un producto de software a lo largo de todo su ciclo de vida. Financian el producto, lo planifican, lo desarrollan, lo gestionan, lo prueban, lo usan, y se benefician con él. Por lo tanto el proceso que conduce este desarrollo debe estar orientado a las personas, es decir, debe funcionar correctamente para la gente que lo usa.

La forma en que se organiza y maneja un proyecto de software afecta profundamente a las personas involucradas en él. Por lo tanto, al considerarse cuestiones clave como la factibilidad del proyecto, la gestión de riesgos, la estructura de los equipos, la planificación del proyecto, su comprensibilidad, y el sentido del cumplimiento, no deben perderse de vista los intereses de las personas.

Hay que distinguir entre una *persona* o "recurso humano", y un *trabajador*. La misma persona puede desempeñar el rol de varios trabajadores en un proyecto. Por ejemplo, Carlos puede ser quien especifica un caso de uso en particular, y también quien realiza la ingeniería de cierto componente. De manera similar, un mismo trabajador puede formarse por un conjunto de personas trabajando juntas. Por ejemplo, puede pasar que el trabajador "Arquitecto" sea en realidad un equipo formado por María y Alberto. Cada trabajador tiene un conjunto de responsabilidades, y lleva a cabo un conjunto de actividades en el desarrollo de software.

#### *Proyecto*

Es el elemento organizativo mediante el cual se gestiona el desarrollo del software. El resultado de un proyecto es una nueva versión de un producto.

El equipo que lleva adelante el proyecto, durante todo su ciclo de vida, se ve afectado por el cambio, las iteraciones, y el patrón organizativo dentro del cual se conduce al proyecto. Todo el curso de un proyecto es una continua *secuencia de*

*cambios*. Cada ciclo, cada fase, cada iteración, cambia el sistema de una cosa a otra diferente.

Dentro de cada fase de un ciclo, los trabajadores llevan a cabo las actividades de la fase mediante una serie de *iteraciones*. Cada iteración implementa un conjunto de casos de uso relacionados o reduce ciertos riesgos. Como cada iteración atraviesa al menos cinco flujos de trabajo básicos (requerimientos, análisis, diseño, implementación, y prueba), una iteración puede pensarse como un miniproyecto.

Un proyecto involucra un equipo de personas asignadas para lograr un resultado dentro de ciertas restricciones de tiempo, costo y calidad. Se intenta proveer un patrón dentro del cual los trabajadores ejecuten el proyecto. Este patrón organizativo indica los tipos de trabajadores que el proyecto necesita y los artefactos con los que éste va a trabajar.

***Producto***

Son los *artefactos* que se crean durante el ciclo de vida del proyecto, como modelos, código fuente, ejecutables, y documentación. Es decir que en el contexto del Proceso Unificado, el producto desarrollado es un sistema de software completo, y no sólo el código que se entrega.

Existen dos tipos de artefactos: los de *ingeniería* y los de *gestión*. Los primeros corresponden al código, los diagramas arquitectónicos, los esquemas de las interfaces, etc. Algunos ejemplos de artefactos de gestión son los planes de desarrollo (de entregas, de iteraciones), un plan para la asignación de personas a distintas tareas, la asignación de responsabilidades, y también las especificaciones del ambiente de desarrollo.

Los *modelos* constituyen el tipo más interesante de los artefactos empleados en el Proceso Unificado. Cada modelo brinda una perspectiva diferente del sistema, de modo que las personas interesadas pueden visualizarlo considerando sólo los aspectos que le interesan. La construcción de un sistema es, por lo tanto, un proceso de construcción de modelos que describen todas las perspectivas diferentes del sistema.

***Proceso***

Un proceso de ingeniería de software es una definición del conjunto completo de actividades necesarias para transformar los requerimientos de los usuarios en un producto que los satisfaga. Un proceso es una plantilla para crear proyectos, es decir que cada proyecto es una *instancia* de un proceso.

Un proceso es la definición de un conjunto de actividades, no su ejecución. Se describe al proceso completo en trozos llamados *flujos de trabajo*. Para describir estos flujos se utilizan los diagramas de actividad de UML.

Un proceso de desarrollo de software tiene que ser adaptable y fácil de configurar para cubrir las necesidades reales de un proyecto (u organización) específico. Esta especialización se verá influida por factores organizacionales, de dominio, del ciclo de vida, y técnicos, específicos de la organización y de aplicación a desarrollar.

***Herramientas***

Es el software que se usa para automatizar las actividades definidas en el proceso. Con esto se puede lograr un incremento en la productividad y en la calidad, y una reducción en el tiempo necesario para completar el proyecto.

### 3.2.2. Un Proceso Dirigido por Casos de Uso

Lo primero que deben hacer los desarrolladores es capturar los requerimientos del cliente en la forma de *Casos de Uso*, y confeccionar el modelo de casos de uso. Luego es necesario analizar y diseñar el sistema para cumplir con los casos de uso, creando así primero un modelo de análisis y luego un modelo de diseño y de despliegue (*deployment model*, que define los nodos físicos de computadoras y el mapeo de los componentes de esos nodos); entonces se implementa el sistema en un modelo de implementación, que incluye todo el código, es decir, los componentes. Finalmente, los desarrolladores preparan un modelo de pruebas que les permite verificar que el sistema provee la funcionalidad descripta por los casos de uso.

Los casos de uso han sido adoptados casi universalmente para la captura de requerimientos de los sistemas de software en general, y de los sistemas basados en componentes en particular, pero son mucho más que una herramienta de captura de requerimientos. Los casos de uso conducen todo el proceso de desarrollo. Son la principal entrada para buscar y especificar clases, subsistemas, e interfaces, para encontrar y especificar casos de prueba, y para planificar las iteraciones del desarrollo y la integración del sistema. Para cada iteración, los casos de uso conducen a través del conjunto completo de flujos de trabajo, desde la captura de requerimientos, pasando por el análisis, diseño e implementación, hasta las pruebas, e integra estos diferentes flujos (Figura 3.23).

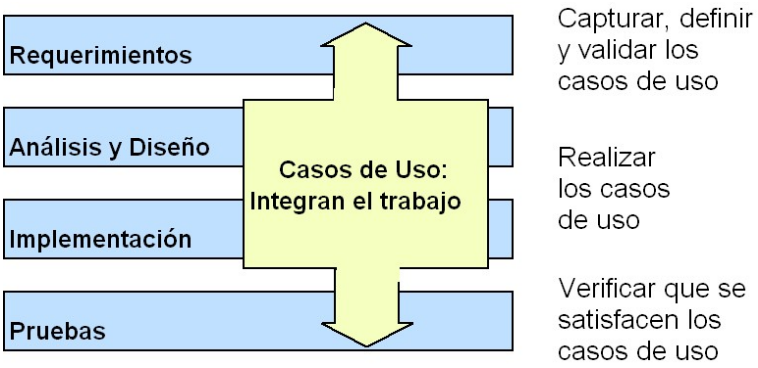


Figura 3.23: Los casos de uso conducen el proceso en todas sus fases

Hay varias razones por las cuales los casos de uso son buenos y se han hecho populares y universalmente adoptados. Las dos razones principales son:

- Ofrecen un medio sistemático e intuitivo para capturar los requerimientos funcionales, poniendo énfasis en el valor agregado al usuario.
- Conducen el proceso de desarrollo en su totalidad, ya que la mayoría de las actividades, como análisis, diseño y prueba, se ejecutan tomando como punto de partida un caso de uso. El diseño y las pruebas también pueden planificarse y coordinarse en términos de casos de uso. Esta característica se hace aún más obvia en el proyecto cuando la arquitectura se ha estabilizado luego de las primeras iteraciones.

Los casos de uso nos ayudan a llevar adelante el desarrollo iterativo. Conducen cada iteración a través de todos los flujos de trabajo, y dan como resultado un *incremento*. Cada incremento del desarrollo es entonces una realización operativa de un conjunto de casos de uso. O dicho en otras palabras, en cada iteración se identifican e implementan unos cuantos casos de uso.

Los casos de uso también son útiles para idear la arquitectura. Mediante la selección de un conjunto adecuado de casos de uso –los más significativos desde el punto de vista arquitectónico– para realizar durante las primeras iteraciones, se puede implementar un sistema con una arquitectura estable, que pueda usarse en muchos ciclos de desarrollo posteriores.

Los casos de uso también se toman como base para la escritura del manual del usuario. Como cada caso de uso describe una forma de usar el sistema, son un punto de partida ideal para explicarle al usuario cómo puede interactuar con el sistema.

### Captura de los Casos de Uso

Definición: *un caso de uso especifica una secuencia de acciones, incluyendo variantes, que el sistema puede ejecutar y que obtiene un resultado observable de valor para un actor particular* [JBR99].

Durante el flujo de trabajo de captura de requerimientos se identifican las necesidades de usuarios y clientes. Los requerimientos funcionales se expresan como casos de uso en un *modelo de casos de uso*, mientras que los demás requerimientos pueden adjuntarse a los casos de uso o mantenerse en forma separada y describirse de alguna otra manera.

El modelo de casos de uso representa, entonces, los requerimientos funcionales del sistema, y ayuda a clientes, usuarios, y desarrolladores, a ponerse de acuerdo en cómo usar el sistema. Cada tipo de usuario se representa como un *actor*. Un *diagrama de casos de uso* describe parte del modelo de casos de uso, y muestra un conjunto de casos de uso y actores, con una asociación entre cada par de ellos que interactúa.

---

**Ejemplo: Un Modelo de Casos de Uso para el Sistema de Cajero Automático**

El Cliente del Banco usa un Sistema de Cajero Automático (SCA) para retirar y depositar dinero y para transferir dinero entre cuentas. Esto se representa en los tres casos de uso de la Figura 3.24 que tienen asociaciones al actor para indicar que interactúan.

---

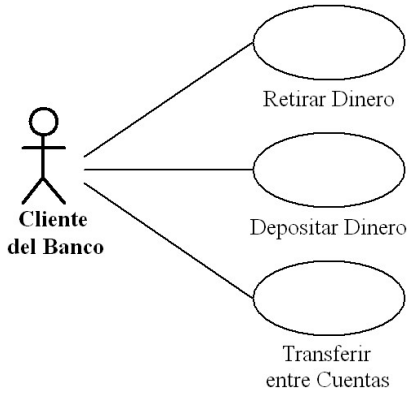


Figura 3.24: Ejemplo de diagrama de casos de uso con un actor y tres casos de uso

Los actores forman el entorno del sistema, y no tienen que representar necesariamente a seres humanos, sino que pueden ser otros sistemas o hardware externo.

Además, un usuario físico puede actuar como uno o varios actores, y varios usuarios individuales pueden actuar como diferentes ocurrencias de un mismo actor.

Los actores y el sistema se comunican entre sí intercambiando mensajes mientras ejecutan un caso de uso. Mediante una buena definición de qué es lo que hacen los actores y qué es lo que hacen los casos de uso, se puede hacer una clara separación entre las responsabilidades de los actores y las del sistema.

Como ya se ha dicho, los casos de uso especifican el sistema, en el sentido de que el modelo de casos de uso captura todos los requerimientos funcionales del mismo. Los casos de uso se pueden encontrar observando la forma en que los usuarios necesitan utilizar el sistema para hacer su trabajo.

La secuencia de acciones ejecutadas por un caso de uso durante su operación (es decir, una instancia del caso de uso) es un camino específico a través del caso de uso. Existen muchos caminos posibles, y muchos pueden ser variantes muy similares de la secuencia de acciones especificadas en el caso de uso. Estas variantes deberían agruparse bajo el mismo caso de uso.

---

**Ejemplo: El Caso de Uso Retirar Dinero**

La secuencia simplificada de acciones para un camino a través de este caso de uso es:

1. El Cliente se identifica.
2. El Cliente elige de qué cuenta retirar el dinero y especifica el monto a retirar.
3. El Sistema deduce el monto de la cuenta y entrega el dinero.

---

Otra variante: Cajero sin dinero suficiente

1. El Cliente se identifica.
2. El Cliente elige de qué cuenta retirar el dinero y especifica el monto a retirar.
3. El Sistema informa que no podrá suministrar el monto solicitado por no tener disponibilidad de dinero.

---

Otra variante: Saldo insuficiente en la cuenta del Cliente

1. El Cliente se identifica.

2. El Cliente elige de qué cuenta retirar el dinero y especifica el monto a retirar.

3. El Sistema informa que no podrá suministrar el monto solicitado porque el mismo excede el saldo actual de la cuenta.

---

El modelo de casos de uso es un vehículo para organizar los requerimientos en una forma fácil de manejar. Los clientes y usuarios pueden entenderlo y usarlo para comunicar sus necesidades de forma consistente y sin redundancias. Los desarrolladores pueden dividir el trabajo de captura de requerimientos entre ellos, y luego usar los resultados (casos de uso) como entradas para analizar, diseñar, implementar, y probar el sistema.

### Análisis, Diseño, e Implementación para Realizar los Casos de Uso

El modelo de análisis se va incrementando a medida que se analizan más y más casos de uso. En cada iteración se elige un conjunto de casos de uso que se realizan en el modelo de análisis. Se construye el sistema como una estructura de clasificadores (clases de análisis) y relaciones entre ellos. También se describe las colaboraciones que realizan los casos de uso. Es esencial que en las primeras iteraciones se identifiquen y elijan los casos de uso más importantes, para así construir una arquitectura estable en las etapas tempranas del sistema.

A la relación que existe entre cada caso de uso y la respectiva colaboración en el modelo de análisis (y viceversa) se la llama dependencia de traza o simplemente *traza*. Existen trazas además entre colaboraciones del modelo de diseño y colaboraciones del modelo de análisis, y entre componentes del modelo de implementación y subsistemas del modelo de diseño. Esta relación no agrega información semántica, sino que sólo conecta los modelos. Sin embargo, es importante saber aplicarlas para agregar comprensión y al tener en cuenta la propagación de cambios.

---

**Ejemplo: La Realización de un Caso de Uso en el Modelo de Análisis**

La Figura 3.25 muestra cómo se realiza el caso de uso *Retirar Dinero*, mediante una colaboración con una dependencia de ≪traza≫ entre ellos, y las cuatro clases de análisis que participan y cumplen un rol en la realización del caso de uso. *Boca de Abastecimiento* e *Interfaz del Cajero* son clases de frontera, *Extracción* es una clase de control, y *Cuenta*

es una clase entidad. Como puede verse en la figura, la notación para una realización de un caso de uso o colaboración es una elipse con línea de puntos.

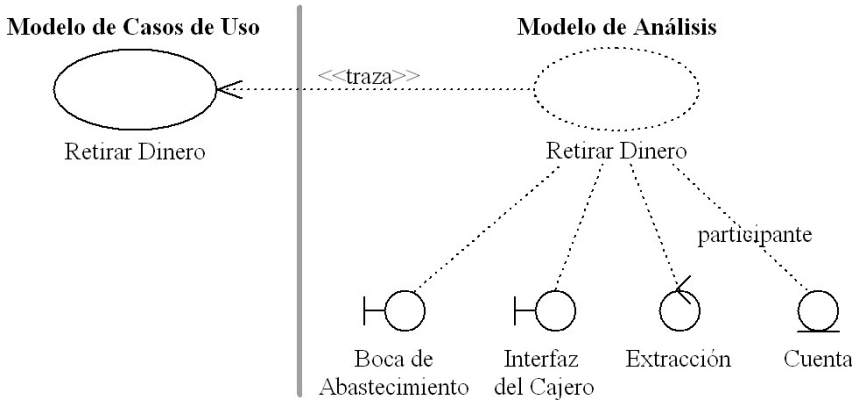


Figura 3.25: Realización del caso de uso Retirar Dinero

En general se comienza con unos pocos casos de uso, creando sus realizaciones e identificando roles para los clasificadores. Luego se hace lo mismo con algunos casos de uso más y se sugieren nuevos roles. Algunos de estos nuevos roles pueden definirse como pertenecientes a clasificadores ya identificados, es decir que cada clasificador puede participar y desempeñar roles en varias realizaciones de casos de uso.

---

**Ejemplo: Una Clase que participa en varias Realizaciones de Casos de Uso en el Modelo de Análisis.**

En el lado izquierdo de la Figura 3.26 se presenta nuevamente el conjunto de casos de uso para el SCA, y en el lado derecho está la correspondiente estructura del sistema, en este caso, las clases de análisis que realizan los casos de uso. La estructura del sistema se modela con un diagrama de clases. Aquí se han usado diferentes sombreados para indicar en qué realizaciones de casos de uso participa cada clase.

El diagrama de clases para el SCA se encontró recorriendo las descripciones de los tres casos de uso y luego buscando las formas de realizar cada uno de ellos. Se podría haber hecho algo como lo siguiente:

- La realización de los tres casos de uso, *Retirar Dinero*, *Transferir entre Cuentas*, y *Depositar Dinero* involucran la clase de frontera *Interfaz del Cajero* y la clase entidad *Cuenta*. La ejecución de cada caso de uso comienza con un objeto de la

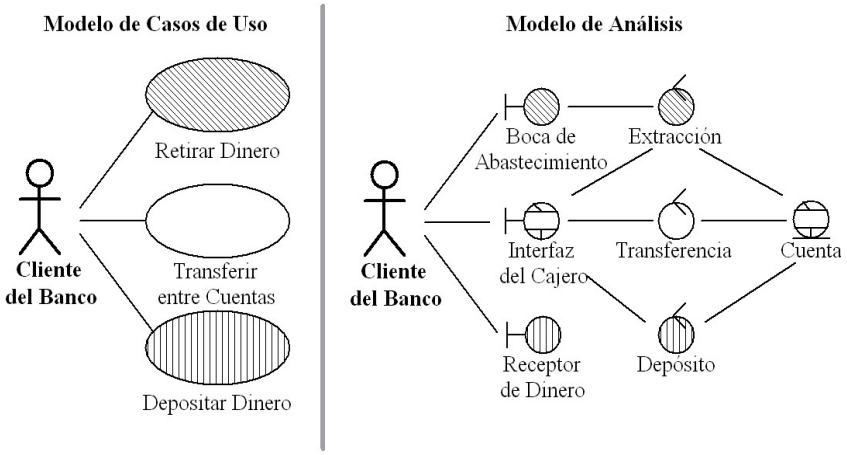


Figura 3.26: Cada caso de uso se realiza como una estructura de clases de análisis

*Interfaz del Cajero*. Luego el trabajo se pasa a un objeto de control que coordina gran parte del caso de uso en cuestión. La clase de este objeto es única para cada caso de uso. La clase *Extracción* participa entonces del caso de uso *Retirar Dinero*, y así sucesivamente. El objeto *Extracción* le ordena a la *Boca de Abastecimiento* entregar el dinero, y al objeto *Cuenta* que reduzca el saldo.

- El objeto *Transferencia* solicita a los dos objetos *Cuenta* involucrados en la realización del caso de uso *Transferir entre Cuentas* que actualicen sus saldos.

- El objeto *Depósito* acepta dinero a través del *Receptor de Dinero* y le pide al objeto *Cuenta* que incremente su saldo.

---

Hasta aquí se ha trabajado para encontrar una estructura estable para el sistema en la iteración actual. Se han identificado las responsabilidades de los clasificadores participantes, y las relaciones entre estos clasificadores. Sin embargo, no se identificó en detalle la interacción que debe tener lugar en la realización de los casos de uso. Se encontró la estructura, pero ahora se necesita superponer a esa estructura los diferentes patrones de interacción requeridos por cada realización de los casos de uso. Es decir, se debe describir cómo se desarrolla o ejecuta (o instancia) una realización de un caso de uso.

Para mostrar cómo deberían interactuar los objetos para ejecutar la realización de un caso de uso, se recurre a los diagramas de colaboración. Un diagrama de colaboración se asemeja a un diagrama de clases, pero en lugar de clases y asociaciones muestra

instancias y enlaces. Representa cómo interactúan los objetos, secuencialmente o en paralelo, mediante la asignación de números a los mensajes intercambiados.

---

**Ejemplo: Utilización de un Diagrama de Colaboración para describir una Realización de un Caso de Uso en el Modelo de Análisis.**

En la Figura 3.27 se usa un diagrama de colaboración para describir cómo se ejecuta el caso de uso *Retirar Dinero* mediante una sociedad de objetos de análisis. El diagrama muestra cómo se mueve el foco de un objeto a otro a medida que el caso de uso se ejecuta, y también los mensajes enviados entre los objetos. Un mensaje enviado por un objeto provoca que el receptor adquiera el foco y ejecute una de las responsabilidades de su clase.

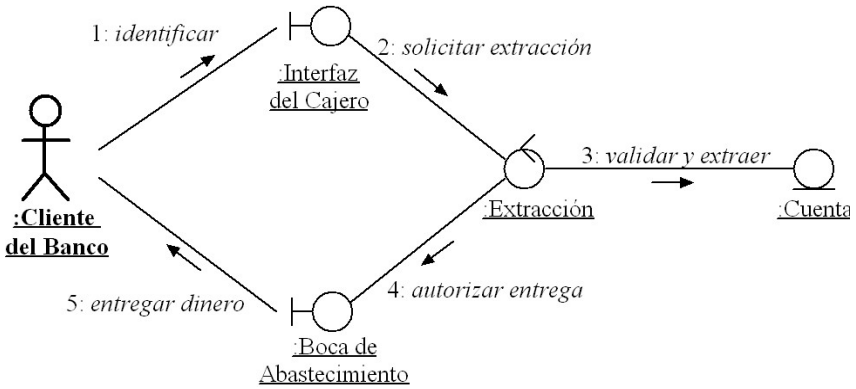


Figura 3.27: Diagrama de colaboración que realiza el caso de uso Retirar Dinero

El nombre de un mensaje denota la intención del objeto emisor al interactuar con el objeto invocado. Más tarde, durante el diseño, estos mensajes se refinan en una o más operaciones provistas por las correspondientes clases de diseño.

---

Como complemento al diagrama de colaboración, los desarrolladores también pueden usar texto en lenguaje natural, texto estructurado, o seudocódigo, para explicar cómo interactúan los objetos para ejecutar el flujo de eventos del caso de uso.

Una vez que se han analizado cada uno de los casos de uso y se han identificado así todos los roles de las clases participantes en cada realización de casos de uso, hay que considerar cómo analizar cada clase. Las responsabilidades de una clase son simplemente una recopilación de todos los roles que juega en todas las realizaciones de casos de uso. Si se reúnen y se eliminan las superposiciones entre los roles, se obtendrá una especificación de todas las responsabilidades y atributos de la clase.

El modelo de diseño se crea usando el modelo de análisis como principal entrada, pero debe adaptarse al ambiente de implementación elegido, y para reusar sistemas heredados. Así, mientras el modelo de análisis trabaja como un primer corte sobre el modelo de diseño, el modelo de diseño trabaja como un anteproyecto para la implementación.

Como en el modelo de análisis, el modelo de diseño también define clasificadores (clases, subsistemas, e interfaces), relaciones entre ellos, y colaboraciones que realizan los casos de uso. Sin embargo, los elementos definidos en el modelo de diseño son la "contraparte de diseño" de los elementos más conceptuales definidos en el modelo de análisis, en el sentido de que los elementos de diseño están adaptados al ambiente de implementación mientras que los elementos de análisis no. Una realización de un caso de uso en el modelo de análisis puede trazarse desde una realización en el modelo de diseño.

---

**Ejemplo: Realizaciones de Casos de Uso en los Modelos de Análisis y de Diseño.**

En la Figura 3.28 se describe cómo el caso de uso *Retirar Dinero* se ha hecho efectivo mediante una realización de caso de uso tanto en el modelo de análisis como en el de diseño.

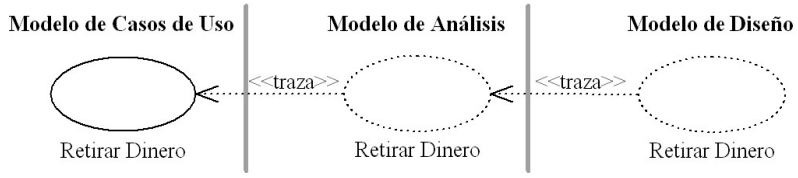


Figura 3.28: Realizaciones de casos de uso en los diferentes modelos

Las realizaciones de casos de uso en los distintos modelos tienen diferentes propósitos. Las clases de análisis *Interfaz del Cajero*, *Extracción*, *Cuenta*, y *Boca de Abastecimiento* participan en la realización del caso de uso *Retirar Dinero* del modelo de análisis. Sin embargo, cuando estas clases de análisis se diseñan, todas especifican y dan lugar a clases de diseño más refinadas que se adaptan al ambiente de implementación, como se ejemplifica en la Figura 3.29.

Por ejemplo, la clase de análisis llamada *Interfaz del Cajero* se diseña mediante cuatro clases de diseño: *Pantalla*, *Teclado*, *Lector de Tarjetas* y *Gestor de Clientes*, que es una clase activa y por lo tanto se representa con borde grueso.

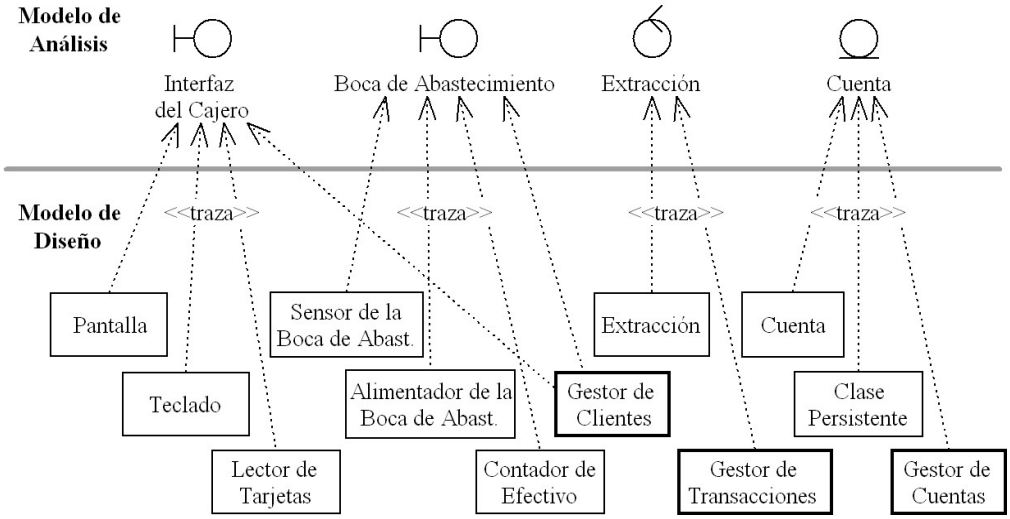


Figura 3.29: Clases de diseño que trazan a clases de análisis

---

Una *clase activa* es una clase cuyas instancias son objetos activos. Un *objeto activo* es un objeto que posee un proceso o hilo y puede iniciar actividad de control.

---

Es normal que la mayoría de las clases de diseño que son específicas de la aplicación tengan una dependencia de traza a sólo una de las clases de análisis. Por lo tanto, la estructura del sistema definida por el modelo de análisis se mantiene naturalmente durante el diseño. Además, las clases activas representan procesos que organizan el trabajo de las demás clases cuando el sistema está distribuido.

En consecuencia, la realización del caso de uso *Retirar Dinero* en el modelo de diseño necesita describir cómo se realiza el caso de uso en términos de las clases de diseño correspondientes. La Figura 3.30 representa un diagrama de clases que es parte de la realización de dicho caso de uso. Es obvio que este diagrama de clases introduce más detalle que el diagrama de clases del modelo de análisis, debido a la adaptación del modelo de diseño al ambiente de implementación.

Como en el caso del análisis, se debe identificar detalladamente la interacción entre los objetos de diseño cuando se realiza el caso de uso en el modelo de diseño. Para modelar estas interacciones entre los objetos en el diseño, se usan principalmente los diagramas de secuencia, como muestra la Figura 3.31. El diagrama de secuencia ilustra cómo se mueve el foco de un objeto a otro a medida que el caso de uso avanza y los mensajes se

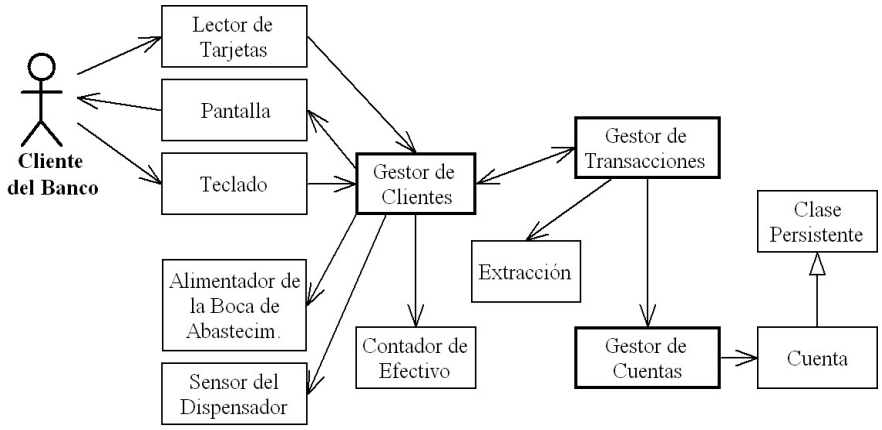


Figura 3.30: Diagrama de clases: parte de la realización de Retirar Dinero

intercambian entre los objetos.

Es interesante notar que todos los mensajes del diagrama de secuencia de la Figura 3.31 se corresponden con los dos primeros mensajes del diagrama de colaboración de la Figura 3.27 ("1: identificar" y "2: solicitar extracción"). Esto nos da una idea de la complejidad y el nivel de detalle introducido en el modelo de diseño en comparación con el modelo de análisis. Aquí también se pueden usar descripciones textuales para complementar los diagramas de secuencia.

---

En un sistema grande con cientos o miles de clases, sería imposible usar sólo clases para realizar los casos de uso: el sistema es demasiado grande como para poder abarcarlo sin una forma de agrupamiento de mayor orden. Las clases se agrupan entonces en *subsistemas*. Un subsistema es un grupo semánticamente útil de clases o de otros subsistemas. Un subsistema tiene un conjunto de *interfaces* que provee y usa. Estas interfaces definen el contexto del subsistema (actores y otros subsistemas y clases).

Los subsistemas de menor nivel se denominan subsistemas de servicio, porque sus clases realizan un servicio. Este tipo de subsistemas conforman una unidad manejable de funcionalidad opcional (o potencialmente opcional).

---

**Ejemplo: Los Subsistemas agrupan Clases.**

Los desarrolladores agrupan las clases en los tres subsistemas que se muestran en la Figura 3.32. Estos subsistemas se eligen de forma que todas las clases que proveen la

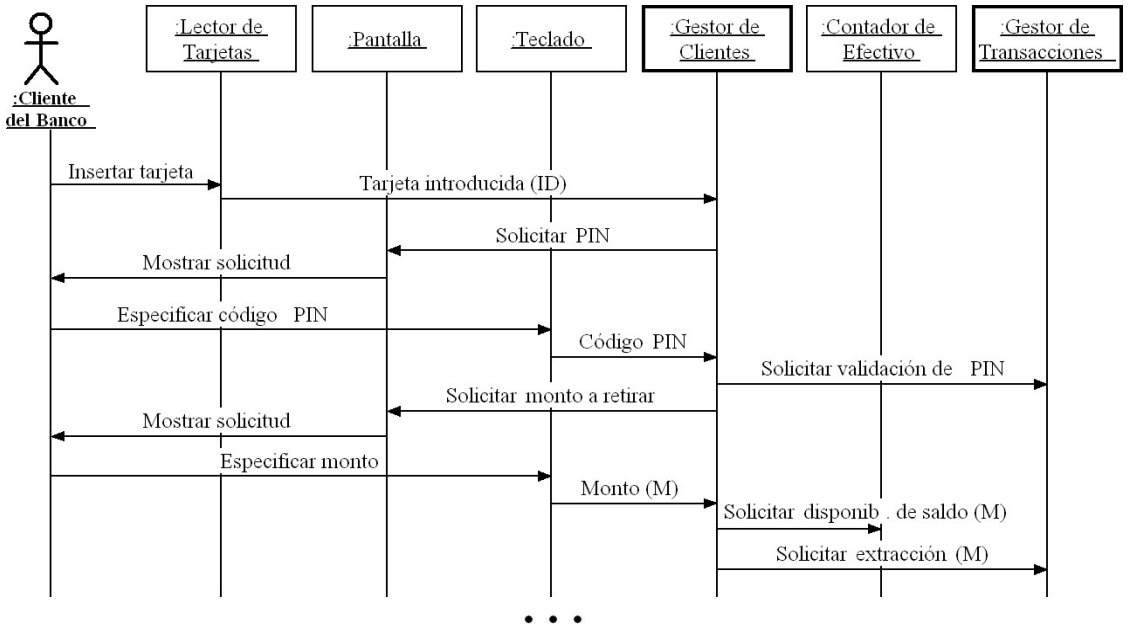


Figura 3.31: Diagrama de secuencia: parte de la realización de Retirar Dinero

interfaz del usuario están ubicadas en un subsistema, todas las que tienen que ver con las cuentas en otro subsistema, y las clases específicas del caso de uso en otro subsistema. Cada una de las clases específicas del caso de uso, como la clase *Extracción*, en el subsistema *Gestión de Transacciones*, termina en un subsistema de servicio separado.

La figura también muestra las interfaces entre los subsistemas. Un círculo representa una interfaz. La línea sólida de una clase a una interfaz significa que la clase provee esa interfaz. Una línea de puntos de una clase a una interfaz significa que la clase usa la interfaz. La interfaz *Transfiere* define operaciones para mover dinero entre cuentas, retirar y depositar dinero. La interfaz *Extracción* define operaciones para solicitar extracciones de una cuenta. La interfaz *Entrega* define operaciones que serán usadas por otros subsistemas, como el subsistema de *Gestión de Transacciones*, para entregar dinero al cliente del banco.

---

Durante el flujo de trabajo de implementación se desarrollan todos los *componentes* necesarios para producir un sistema ejecutable: componentes ejecutables, archivos, tablas, etc. Un componente es una parte física y reemplazable del sistema que provee la realización a un conjunto de interfaces. El modelo de implementación está hecho de componentes, que incluyen todos los ejecutables, como así también todos los demás tipos de componentes.

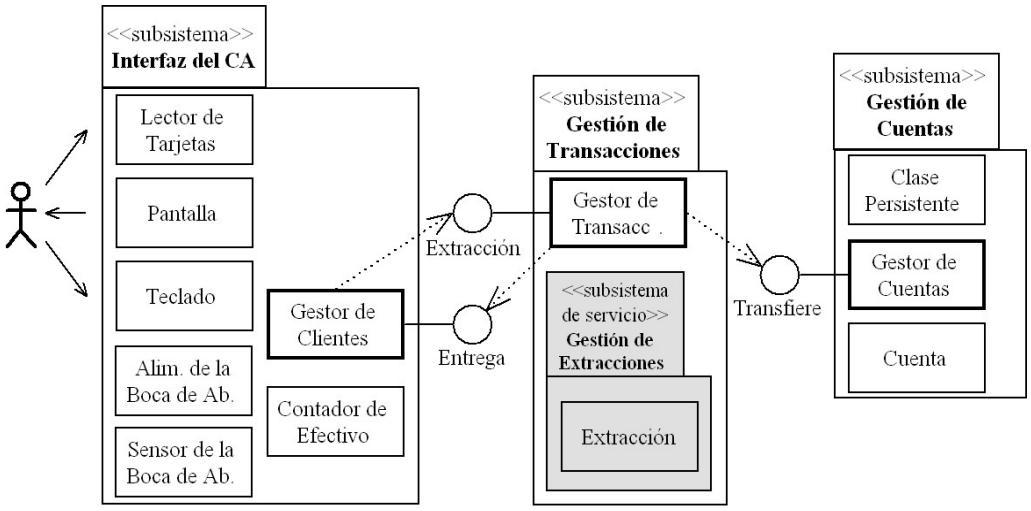


Figura 3.32: Tres subsistemas y un subsistema de servicio para el ejemplo

---

**Ejemplo: Componentes en el Modelo de Implementación**

La Figura 3.33 representa los componentes que implementan las clases de diseño presentadas anteriormente en el modelo de diseño.

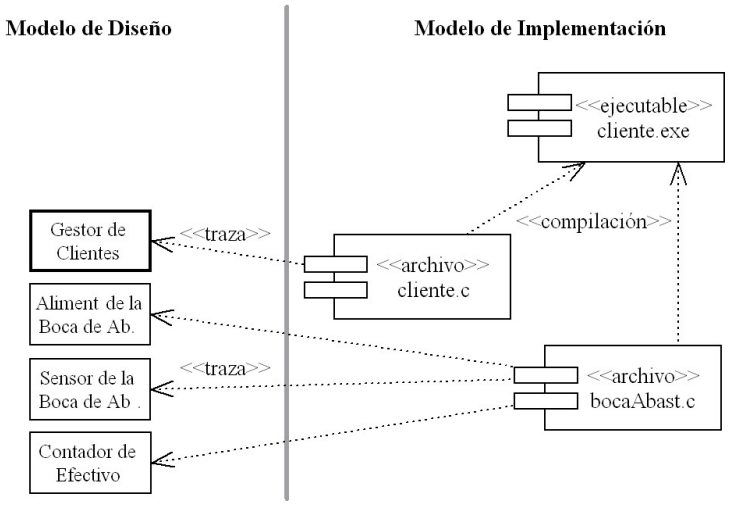


Figura 3.33: Componentes que implementan las clases de diseño

Por ejemplo, el componente *bocaAbast.c* contiene el código fuente (y por lo tanto implementa) las tres clases, *Alimentador de la Boca de Abastecimiento*, *Sensor de la Boca de Abastecimiento*, y *Contador de Efectivo*. Luego, este componente archivo se

compila y vincula junto con el componente archivo *cliente.c* en el componente *cliente.exe*, que es ejecutable.

---

Pero la implementación es más que desarrollar el código para crear un sistema ejecutable. Los desarrolladores responsables de implementar un componente también son responsables de la prueba de unidad antes de pasarlo a las pruebas de integración y del sistema.

### Prueba de los casos de uso

Durante las pruebas, se verifica que el sistema implementa correctamente su especificación. Se desarrolla un modelo de prueba que consiste en casos de prueba y procedimientos de prueba, y luego se ejecutan los procedimientos de prueba para asegurarse de que el sistema trabaja como se esperaba. Un caso de prueba es un conjunto de entradas de prueba, condiciones de ejecución, y resultados esperados desarrollados con un objetivo en particular, como ejecutar un camino específico a través de un caso de uso o verificar la conformidad con un requerimiento determinado. Un procedimiento de prueba es una especificación de cómo realizar la disposición, ejecución, y evaluación de resultados para un caso de prueba en particular. Los defectos hallados se analizan para localizar el problema. Luego estos problemas se ordenan por prioridad y se corrigen.

Este enfoque, en cierto sentido, es nuevo, ya que los casos de prueba se identifican incluso antes de comenzar con el diseño del sistema, y así se puede asegurar que el diseño implementa realmente los casos de uso.

---

**Ejemplo: Identificación de un Caso de Prueba a partir de un Caso de Uso.**

La Figura 3.34 representa un caso de prueba, *Retirar Dinero–Flujo Básico*, que especifica cómo probar el flujo básico del caso de uso *Retirar Dinero*.

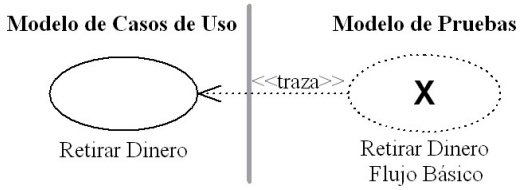


Figura 3.34: Caso de prueba para el caso de uso Retirar Dinero

Se ha introducido un nuevo estereotipo para los casos de prueba, un símbolo de caso de uso con una cruz en su interior. El caso de prueba especifica la entrada, resultados esperados, y otras condiciones relevantes para verificar el flujo básico del caso de uso *Retirar Dinero*:

**Entradas:**

- La cuenta 12-121-1211 del Cliente tiene un saldo de $350.
- El Cliente se identifica correctamente.
- El Cliente solicita retirar $200 de la cuenta 12-121-1211.
- Hay suficiente dinero (al menos $200) en el Cajero Automático.

**Resultado:**

- El saldo de la cuenta 12-121-1211 del Cliente disminuye a $150.
- El Cliente recibe $200 del Cajero Automático.

**Condiciones:** Durante este caso de prueba no se le permite a ningún otro caso de uso (instancia) acceder a la cuenta 12-121-1211.

---

Al identificar tempranamente los casos de uso, se pueden comenzar a planear las actividades de prueba también en forma temprana, y se pueden sugerir casos de prueba útiles. Estos casos de prueba pueden luego hacerse más elaborados durante el diseño, cuando se conoce mejor la forma en que el sistema va a ejecutar los casos de uso.

### 3.2.3. Un Proceso Centrado en la Arquitectura

Los casos de uso no son suficientes para lograr un sistema funcional. Se necesita algo más, y ese "algo más" es la *arquitectura*. Se puede pensar la arquitectura de un sistema como la visión común sobre la que deben ponerse de acuerdo, o al menos aceptar, todos los trabajadores (desarrolladores y otros interesados). La arquitectura brinda una perspectiva clara del sistema completo, lo que es indispensable para controlar su desarrollo. Algunos de los trabajos más importantes sobre Arquitectura de Software que pueden mencionarse son [Bus93], [G+93], [WN95], [M+97] y [BCK03].

Se necesita una arquitectura que describa los elementos más importantes del modelo. Su importancia radica en el hecho de que sirven de guía en el trabajo con el sistema, tanto en el ciclo actual como a través de todo el ciclo de vida. Estos elementos arquitectónicamente significativos incluyen algunos de los subsistemas, dependencias, interfaces, colaboraciones, nodos, y clases activas. Describen la subestructura del sistema que se necesita como base para comprenderlo, desarrollarlo, y evolucionarlo en forma efectiva.

El desarrollo de la mayoría de los sistemas de software, de manera similar a lo que ocurre con la construcción de un edificio, implica previsión y el registro de esas intenciones en una forma útil no sólo para los desarrolladores siguientes, sino también por los demás interesados. Más aún, estas previsiones, esta arquitectura, no surgen de la nada. Los arquitectos las desarrollan a lo largo de varias iteraciones durante las fases de inicio y de elaboración. De hecho, el objetivo primario de la fase de elaboración es establecer una sólida arquitectura ejecutable. Como resultado, se ingresa a la fase de construcción con una base firme sobre la cual erigir el sistema completo.

Un sistema de software es una entidad única, pero el arquitecto de software y los desarrolladores encuentran útil presentar al sistema desde diferentes perspectivas para comprender mejor su diseño. Estas perspectivas se denominan *vistas* de los modelos del sistema. Juntas, las vistas presentan la arquitectura.

La arquitectura del sistema comprende las decisiones significativas acerca de:

- La organización del sistema de software.
- Los elementos estructurales, y sus interfaces, que abarcará el sistema, junto con su comportamiento, especificado en las colaboraciones entre esos elementos.
- La composición de los elementos estructurales y de comportamiento en subsistemas cada vez más grandes.
- El estilo arquitectónico que guía a la organización.

### ¿Por qué se necesita una arquitectura?

Un sistema de software grande y complejo requiere un arquitecto, de forma que los desarrolladores puedan progresar hacia una visión común. Un sistema de software es difícil de visualizar debido a que no existe en nuestro mundo de tres dimensiones. Con

frecuencia, en algún aspecto es único o sin precedentes. A veces usa tecnología aún no probada, o una mezcla de tecnologías novedosas. También puede llevar hasta sus límites a las tecnologías existentes. Además, debe construirse de modo que se pueda adaptar a una enorme variedad de cambios futuros. A medida que el sistema se vuelve más complejo, el problema de diseño va más allá de los algoritmos y estructuras de datos de la computación: el diseño y la especificación de la estructura global del sistema surge como un nuevo tipo de problema.

Además, con frecuencia se tiene un sistema preexistente que ejecuta algunas de las funciones del sistema propuesto. Suponer qué hace ese sistema, a menudo con escasa o ninguna documentación, y qué código podrían reusar los desarrolladores, agrega complejidad al desarrollo.

Entonces, se necesita una arquitectura para

- Comprender el sistema.
- Organizar el desarrollo.
- Propiciar el reuso.
- Evolucionar el sistema.

### Casos de Uso y Arquitectura

Ya se ha mencionado que hay cierta interacción entre casos de uso y arquitectura, y se mostró cómo desarrollar un sistema que ofrezca los casos de uso correctos a sus usuarios. Si el sistema ofrece los casos de uso correctos –casos de uso con desempeño, calidad, y utilidad elevados– entonces los usuarios pueden usarlo para llevar a cabo su misión. Pero para llegar a esto es necesario construir una arquitectura que permita implementar los casos de uso con un bajo costo, ahora y en el futuro.

Como ya se vio, la arquitectura se ve influenciada por los casos de uso que se pretende que el sistema soporte; los casos de uso son *conductores* para la arquitectura. En las primeras iteraciones se seleccionan unos pocos casos de uso que se supone permitirán delinear mejor la arquitectura. Estos casos de uso arquitectónicamente significativos incluyen aquellos que son más necesarios para los clientes en la próxima entrega y quizás en las futuras. Sin embargo, la arquitectura no sólo es influenciada por estos casos de uso, sino también por otros factores, como el tipo de sistema de

software que se quiere construir, los sistemas heredados, los estándares y políticas de la compañía, requerimientos no funcionales y necesidades de distribución.

Una vez que se erigió una arquitectura estable, se puede implementar la funcionalidad completa mediante la realización del resto de los casos de uso durante la fase de construcción. Los casos de uso implementados durante esta etapa se desarrollan en su mayor parte usando como entrada los requerimientos de clientes y usuarios. Pero los casos de uso también se ven influenciados por la arquitectura elegida en la etapa de elaboración.

A medida que se capturan nuevos casos de uso, se usan los conocimientos de la arquitectura ya construida para realizar mejor el trabajo. Cuando se evalúan el costo y valor de cada caso de uso sugerido, se lo hace a la luz de la arquitectura existente. Algunos casos de uso serán fáciles de implementar, mientras que otros serán más difíciles. Se negocia con el cliente y se decide si se podrían modificar los casos de uso, haciéndolos más acordes con la arquitectura ya construida, para simplificar la implementación. Así, con lo que ya se tiene, se pueden crear nuevos casos de uso, subsistemas, y clases con un bajo costo.

Entonces, por un lado, los casos de uso conducen a la arquitectura. Por el otro lado, usamos nuestro conocimiento de la arquitectura para hacer mejor nuestro trabajo cuando se capturan los requerimientos como casos de uso. La arquitectura *guía* a los casos de uso (Figura 3.35).

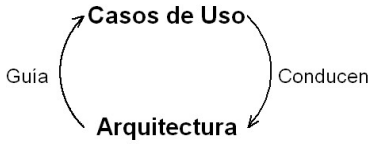


Figura 3.35: Interrelación entre casos de uso y arquitectura

Para resolver este problema "del huevo y la gallina", se recurre a la *iteración*. Primero, se construye una arquitectura tentativa en base a una buena comprensión del área de dominio pero sin considerar los casos de uso detallados. Luego se toman unos pocos casos de uso significativos y se mejora la arquitectura adaptándola para soportar esos casos de uso. Luego se toman algunos casos de uso más y se construye una arquitectura aún mejor, y así sucesivamente. En cada iteración se toma e implementa un conjunto de casos de uso para validar y, si es necesario, mejorar la arquitectura. En

cada iteración también se implementa un poco más de las partes de la arquitectura específicas de la aplicación, en base a los casos de uso elegidos. Así los casos de uso nos ayudan a mejorar gradualmente la arquitectura a medida que iteramos hacia un sistema completo.

### Una Descripción de la Arquitectura

La descripción de la arquitectura no es más que un extracto apropiado de los modelos del sistema (es decir, no agrega nada nuevo). La primera versión de la descripción de la arquitectura es un extracto de la versión de los modelos que se tienen al final de la fase de elaboración en el primer ciclo de vida. Dado que no se trata de hacer una reescritura más legible de esos extractos, la descripción de la arquitectura se parece mucho a los modelos ordinarios del sistema. Esta semejanza significa que la vista arquitectónica del modelo de casos de uso parece un modelo de casos de uso común. La única diferencia es que la vista arquitectónica sólo contiene casos de uso arquitectónicamente significativos, mientras que el modelo de casos de uso final contiene todos los casos de uso. Lo mismo se aplica para la vista arquitectónica del modelo de diseño. Se parece a un modelo de diseño, pero sólo realiza los casos de uso arquitectónicamente interesantes. Se seguirá usando el sencillo ejemplo del *Sistema de Cajero Automático* para ilustrar qué deben contener las vistas arquitectónicas, comparando lo que debe haber en las vistas y en los modelos completos del sistema.

La descripción de la arquitectura tiene cinco secciones, una por cada modelo. Hay una vista del modelo de casos de uso, una vista del modelo de análisis (que no siempre se conserva), una vista del modelo de diseño, una vista del modelo de despliegue, y una vista del modelo de implementación.

#### *La Vista Arquitectónica del Modelo de Casos de Uso*

Esta vista presenta los actores y casos de uso (o escenarios de estos casos de uso) más importantes.

---

**Ejemplo: Vista Arquitectónica del Modelo de Casos de Uso del Sistema de CA.**

En el ejemplo del CA, *Retirar Dinero* es el caso de uso más importante. Sin él, no habría un sistema de CA real. Los casos de uso *Depositar Dinero* y *Transferir entre Cuentas* parecen menos importantes para el cliente de banco medio.

Por lo tanto, para definir la arquitectura, el arquitecto sugiere implementar completamente el caso de uso *Retirar Dinero* durante la fase de elaboración. Ningún otro caso de uso (o parte) se estima interesante para propósitos arquitectónicos. Así, la vista arquitectónica del modelo de casos de uso debería mostrar la descripción completa del caso de uso *Retirar Dinero*.

---

***La Vista Arquitectónica del Modelo de Diseño***

Esta vista presenta los clasificadores arquitectónicamente más interesantes del modelo de diseño: los subsistemas e interfaces más importantes, como así también unas pocas clases muy importantes, principalmente las clases activas. También presenta cómo se realizan los casos de uso más importantes en términos de esos clasificadores, es decir como realizaciones de casos de uso.

---

**Ejemplo: Vista Arquitectónica del Modelo de Diseño del Sistema de CA**

Se habían identificado tres clases activas: *Gestor de Clientes*, *Gestor de Transacciones*, y *Gestor de Cuentas* (Figura 3.36). Estas clases activas se incluyen en la vista arquitectónica del modelo de diseño.

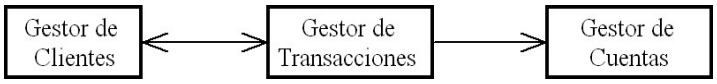


Figura 3.36: Estructura estática: clases activas

Además, se sabe que existen tres subsistemas: *Interfaz del CA*, *Gestión de Transacciones*, y *Gestión de Cuentas* (Figura 3.37). Estos subsistemas son necesarios para realizar el caso de uso *Retirar Dinero*, por lo tanto son los subsistemas arquitectónicamente significativos. El modelo de diseño incluye otros subsistemas, pero no se consideran aquí.

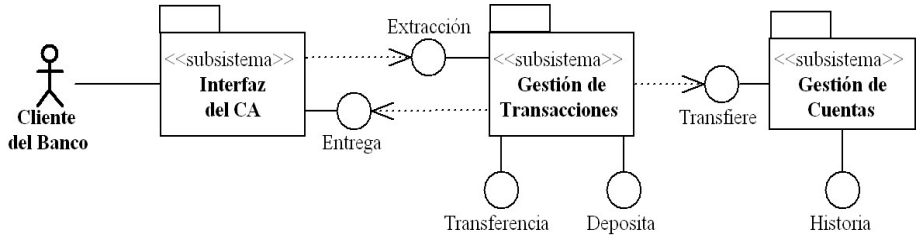


Figura 3.37: Estructura estática: subsistemas e interfaces entre ellos

El subsistema *Interfaz del CA* maneja todas las entradas desde y salidas hacia el cliente del banco, como impresiones de recibos y comandos ingresados por el cliente. El subsistema *Gestión de Cuentas* mantiene toda la información persistente de las cuentas y se usa para todas las transacciones de cuentas. El subsistema *Gestión de Transacciones* contiene las clases para el comportamiento específico de los casos de uso.

Los subsistemas de la Figura 3.37 le proveen comportamiento a los demás mediante interfaces, como la interfaz *Transfiere* provista por *Gestión de Cuentas*. Las interfaces *Transfiere*, *Extracción*, y *Entrega* ya se describieron anteriormente. También están las interfaces *Transferencia*, *Deposita*, e *Historia*, pero estas no se explican porque no están involucradas en el caso de uso que se discute en este ejemplo.

La estructura estática no es suficiente. También es necesario mostrar cómo se realizan los casos de uso arquitectónicamente interesantes, mediante los subsistemas del modelo de diseño. Por lo tanto, se describirá una vez más el caso de uso *Retirar Dinero*, esta vez en términos de subsistemas y actores que interactúan, como se muestra en la Figura 3.38 usando un diagrama de colaboraciones. Los objetos de clases pertenecientes a los subsistemas interactúan unas con otras; estos intercambios se muestran en el diagrama. Los mensajes llevan nombres que especifican operaciones pertenecientes a las interfaces de los subsistemas. Esto se indica con la notación :: (por ejemplo, *Extracción::ejecutar(monto, cuenta)*, donde *Extracción* es una interfaz provista por una clase del subsistema *Gestión de Transacciones*).

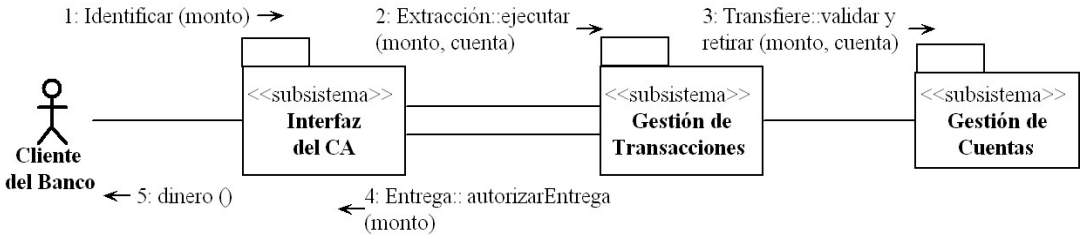


Figura 3.38: Subsistemas que colaboran para ejecutar el caso de uso *Retirar Dinero*

La siguiente lista explica brevemente el flujo en la realización del caso de uso. El texto se presenta aquí en términos de subsistemas, en lugar de clases.

**Precondición:** el cliente del banco tiene una cuenta bancaria que trabaja con el CA:

1. El actor *Cliente del Banco* elige retirar dinero y se identifica a la *Interfaz del CA*, puede ser usando una tarjeta magnética con un número y un PIN. El *Cliente del*

*Banco* también especifica cuánto retirar y de qué cuenta. Aquí se ha asumido que el subsistema *Interfaz del CA* sería capaz de validar la identidad.

2. *Interfaz del CA* solicita al subsistema *Gestión de Transacciones* retirar el dinero. El subsistema *Gestión de Transacciones* es el responsable de ejecutar la secuencia completa de extracción como una transacción atómica, de forma que el dinero sea deducido de la cuenta y entregado al *Cliente del Banco*.

3. *Gestión de Transacciones* solicita al subsistema *Gestión de Cuentas* retirar el dinero. El subsistema *Gestión de Cuentas* determina si puede retirarse el dinero y, en ese caso, deduce la suma de la cuenta y devuelve una respuesta que especifica que es posible ejecutar la extracción.

4. *Gestión de Transacciones* autoriza al subsistema *Interfaz del CA* a entregar el dinero.

5. *Interfaz del CA* entrega el dinero al *Cliente del Banco*.

---

***La Vista Arquitectónica del Modelo de Despliegue***

Este modelo define la arquitectura física del sistema en términos de nodos conectados. Estos nodos son unidades de hardware sobre los que se pueden ejecutar los componentes de software. Con frecuencia sabemos cómo luce la arquitectura física del sistema antes de comenzar a desarrollar el sistema. Los nodos y conexiones pueden entonces modelarse en el modelo de despliegue incluso durante la etapa del análisis de requerimientos.

Durante el diseño, se decide qué clases con activas, es decir, hilos o procesos. Se decide qué debe hacer cada objeto activo, cuál debería ser el ciclo de vida de los objetos activos, y cómo se deberían comunicar, sincronizar, y compartir información. Los objetos activos se ubican en los nodos del modelo de despliegue. Cuando se asignan los objetos activos a los nodos, se considera el potencial de los nodos, como su capacidad de procesamiento y tamaño de memoria, y las características de las conexiones, como su ancho de banda y disponibilidad.

Los nodos y conexiones del modelo de despliegue y la asignación de los objetos activos a los nodos pueden ilustrarse en diagramas de despliegue. Estos diagramas también pueden mostrar cómo se asignan los componentes ejecutables a los nodos. El sistema de CA del ejemplo se distribuye en tres nodos diferentes.

**Ejemplo: Vista Arquitectónica del Modelo de Despliegue del Sistema de CA.**

El *Cliente del Banco* accede al sistema mediante un nodo *Cliente del CA*, que accede al *Servidor de Aplicaciones del CA* para realizar las transacciones (Figura 3.39). El *Servidor de Aplicaciones del CA* usa, a su vez, el *Servidor de Datos del CA* para efectuar transacciones específicas sobre, por ejemplo, cuentas. Esto es así no sólo para el caso de uso *Retirar Dinero*, sino también para los demás casos de uso.

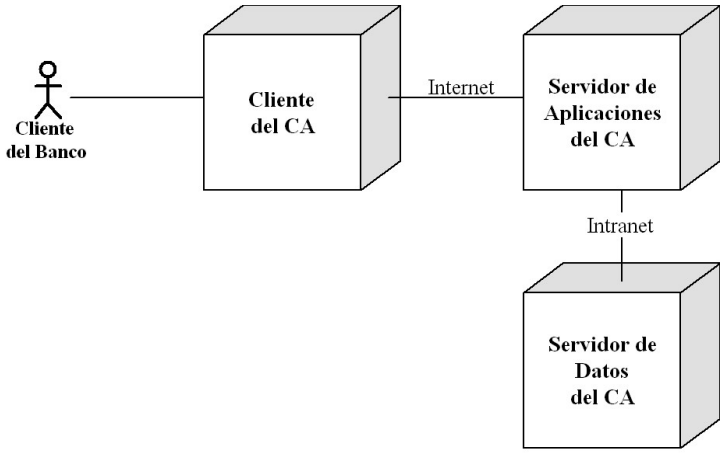


Figura 3.39: Modelo de despliegue con sus tres nodos

Cuando se definen los nodos, es posible distribuir funcionalidad en ellos. Por simplicidad se hace asignando cada subsistema como un todo en un nodo único. El subsistema *Interfaz del CA* se asigna al nodo *Cliente del CA*, el subsistema *Gestión de Transacciones* al *Servidor de Aplicaciones del CA*, y el subsistema *Gestión de Cuentas* al *Servidor de Datos del CA*. Como consecuencia, cada clase activa dentro de esos subsistemas se coloca en el correspondiente nodo, y representa un proceso ejecutándose en el nodo. Cada uno de esos procesos sustenta, y mantiene en su espacio de proceso, objetos de las otras clases (ordinarias, no activas) dentro del subsistema. La distribución de objetos activos, que se muestran como rectángulos con bordes gruesos, se muestra en la Figura 3.40.

### *La Vista Arquitectónica del Modelo de Implementación*

Este modelo es un mapeo directo desde los modelos de diseño y despliegue. Cada subsistema de servicio del diseño generalmente resulta en un componente para cada tipo de nodo sobre los que debe instalarse –aunque no siempre. A veces el mismo componente puede instanciarse y ejecutarse sobre varios nodos. Algunos lenguajes

proveen un constructor para empaquetar componentes. De otro modo, las clases se organizan en archivos de código que representan el conjunto elegido de componentes.


Figura 3.40: Vista arquitectónica del modelo de despliegue

## Resumen

### *¿Qué es Arquitectura?*

Es lo que el arquitecto especifica en una descripción de la arquitectura. La descripción de la arquitectura le permite al arquitecto controlar el desarrollo del sistema desde una perspectiva técnica. La arquitectura del software se enfoca tanto en los elementos estructurales significativos del sistema, como subsistemas, clases, componentes y nodos, como también en las colaboraciones que ocurren entre estos elementos mediante sus interfaces.

Los casos de uso conducen la arquitectura para hacer que el sistema provea el uso y funcionalidad deseados, alcanzando objetivos de desempeño razonables. Una arquitectura tiene que ser integral, pero también tiene que ser lo bastante elástica como para adecuarse a nuevas funciones y tiene que soportar el reuso de software existente.

### *¿Cómo se obtiene?*

La arquitectura se desarrolla iterativamente durante la fase de elaboración a través de los requerimientos, análisis, diseño, implementación, y prueba. Para implementar la arquitectura básica se usan los casos de uso arquitectónicamente significativos y una variedad de otras entradas.

### *¿Cómo se describe?*

La descripción de la arquitectura es una vista de los modelos del sistema, vistas de los modelos de casos de uso, análisis, diseño, implementación, y despliegue. La descripción de la arquitectura detalla las partes del sistema que son importantes y que deben ser comprendidas por todos los desarrolladores y demás interesados.

### 3.2.4. Un Proceso Iterativo e Incremental

Para ser efectivo, un proceso de software debe tener una secuencia de hitos claramente articulados que provean a los gerentes y al resto del equipo los criterios que necesitan para autorizar el paso de una fase a la siguiente dentro del ciclo del producto. Dentro de cada fase, el proceso atraviesa una serie de iteraciones e incrementos que conducen a estos criterios.

En la fase de inicio, el criterio esencial es la *viabilidad*, que se aborda mediante:

- La identificación y reducción de riesgos críticos para la viabilidad del sistema.
- El paso de un subconjunto clave de requerimientos, a través del modelado de los casos de uso, hacia un arquitectura candidata.
- La confección de una estimación inicial de costo, esfuerzo, cronograma, y calidad del producto.
- El inicio del caso de negocio.

En la fase de elaboración, el criterio esencial es la *capacidad de construir el sistema con un trabajo de marco económico*, y se aborda mediante:

- La identificación y reducción de los riesgos que afectan significativamente la construcción del sistema.
- La especificación de la mayoría de los casos de uso que representan la funcionalidad a desarrollar.
- La extensión de la arquitectura candidata a proporciones ejecutables.
- La preparación de un plan de proyecto en suficiente detalle como para guiar la fase de construcción.
- La realización de una estimación, dentro de límites lo suficientemente estrechos como para justificar una propuesta de negocio.
- La finalización del caso de negocio – el proyecto vale la pena.

En la fase de construcción, el criterio esencial es un *sistema capaz de ejecutar las operaciones iniciales dentro del entorno del usuario*, y se trata mediante:

- Una serie de iteraciones, que conducen a construcciones e incrementos periódicos, de forma que a lo largo de toda la fase la viabilidad del sistema es siempre evidente en forma ejecutable.

En la fase de transición, el criterio esencial es un *sistema que logra su capacidad de operación final*, y se aborda mediante:

- La modificación del producto para aliviar problemas no identificados en las fases anteriores.
- La corrección de defectos.

Como ya se vio, el hecho de que un proceso sea dirigido por casos de uso significa que cada fase en el camino hacia un eventual producto se basa en lo que los usuarios realmente hacen. Que sea centrado en la arquitectura significa que el trabajo de desarrollo se concentra, en las primeras fases, en lograr el patrón arquitectónico que guiará la construcción del sistema, asegurando una progresión suave no sólo en la versión actual del producto, sino en la vida completa del mismo.

No es fácil lograr un balance apropiado entre casos de uso y arquitectura (es decir, entre función y forma) de un producto de software. Esto se logra a través del tiempo, mediante una serie de *iteraciones*, por lo que el enfoque de desarrollo iterativo e incremental constituye el tercer aspecto clave del Proceso Unificado.

Este concepto brinda una estrategia para desarrollar un producto de software en pasos pequeños y manejables, es decir: se planea un poquito; se especifica, diseña, e implementa un poquito; y se integra, prueba, y corre un poquito con cada iteración. Si un paso nos deja conformes, se procede con el paso siguiente. En medio de cada paso se obtiene información que permite ajustar el foco en el próximo paso. Luego se ejecuta otro paso, y luego otro. Cuando se han realizado todos los pasos planeados, se tiene un producto desarrollado que se puede entregar a clientes y usuarios.

Un proyecto de desarrollo de software transforma un “delta” (o cambio) en los requerimientos del usuario, en un “delta” (o cambio) en el producto de software. Con un enfoque iterativo e incremental esta adaptación del cambio se hace poco a poco. En otras palabras, se divide el proyecto en un número de *miniproyectos*, donde cada uno es una iteración. Cada iteración tiene todo lo que tiene un proyecto de desarrollo de software: planificación, el paso por una serie de flujos de trabajo (requerimientos, análisis y diseño, implementación, prueba), y preparación para la entrega.

El ciclo de vida iterativo produce resultados tangibles en la forma de *versiones internas* (preliminares), donde cada una agrega un incremento y demuestra la reducción de los riesgos con los que debía tratar. Estas versiones pueden mostrarse a clientes y usuarios, y así producir un retroalimentación valiosa.

En resumen, un ciclo de vida está conformado por una secuencia de iteraciones. Algunas, en particular las primeras, ayudan a comprender los riesgos, establecer la factibilidad, construir el núcleo inicial del software, y realizar el caso de negocio. Otras, particularmente las últimas, agregan incrementos hasta alcanzar un producto listo para ser entregado.

Las iteraciones ayudan a la gerencia a planear, organizar, monitorear, y controlar el proyecto. Las iteraciones se organizan dentro de las cuatro fases, cada una con necesidades particulares de personal, fondos, tiempo, y criterios de entrada y salida. Al comienzo de cada fase, la gerencia puede decidir cómo ejecutarla, qué resultados deben producirse, y qué riesgos deben minimizarse.

### ¿Por qué un desarrollo Iterativo e Incremental?

En dos palabras: mejor software. En unas pocas palabras más, para cumplir con los hitos principales y secundarios con los que se controla el desarrollo. Y además:

- Para manejar tempranamente los riesgos críticos y significativos.
- Para establecer en adelante una arquitectura que guíe el desarrollo del software.
- Para proveer un marco que soporte mejor los cambios inevitables, como los de requerimientos.
- Para construir el sistema en forma incremental, en lugar de hacerlo todo de una vez cerca del final, cuando el cambio se vuelve caro.
- Para proveer un proceso de desarrollo a través del cual el personal pueda trabajar más efectivamente.

### El enfoque iterativo es Conducido por los Riesgos

Un riesgo es una variable del proyecto que compromete o suprime el éxito de un proyecto. Es la probabilidad de que un proyecto experimente eventos indeseables, como demoras en la programación, superación de los costos, o la cancelación absoluta.

Las iteraciones se identifican, priorizan, y ejecutan en base a los riesgos y a su orden de importancia. Esto es así cuando se evalúan nuevas tecnologías, cuando se trabaja para satisfacer las necesidades del cliente, cuando se establece una arquitectura que deberá ser robusta. Es decir que se organizan las iteraciones para lograr la reducción de los riesgos. Esta reducción es central para las iteraciones en las fases de inicio y elaboración. Más tarde, en la fase de construcción, los riesgos han sido en su mayor parte reducidos a un nivel de rutina.

Una vez que los riesgos han sido identificados y priorizados, el equipo debe decidir cómo encarar cada uno. Esencialmente, hay cuatro opciones: evitarlo, limitarlo, mitigarlo, o monitorearlo.

- Algunos riesgos pueden y deben evitarse, quizás replaneando el proyecto o cambiando los requerimientos.
- Otros riesgos deberían limitarse, es decir, restringirse de forma tal que sólo afecten una pequeña parte del proyecto o del sistema.
- Algunos riesgos pueden mitigarse poniéndolos a prueba y viendo si se materializan o desaparecen. Si un riesgo se materializa, el lado positivo es que el equipo ha aprendido más acerca del mismo. El equipo puede entonces estar en condiciones de encontrar una forma de evitar, limitar, o monitorear el riesgo.
- Algunos riesgos, sin embargo, no pueden mitigarse. El equipo sólo puede monitorearlos y ver si se materializan. Si uno de ellos aparece, el equipo tiene que seguir su plan de contingencia. Si surge un riesgo "asesino de proyectos", hay que decidir si se sigue adelante o se cancela el proyecto. En este punto, sólo se ha gastado una cantidad limitada de tiempo y dinero. Sabíamos que podía darse un "asesino de proyectos" –esa es la razón por la que se estaban haciendo esas primeras iteraciones. Por lo tanto, se hizo un buen trabajo al encontrar un riesgo de esta magnitud antes de involucrar a todos los desarrolladores en el proyecto.

Tratar con un riesgo lleva tiempo. Evitarlo o limitarlo implica volver a hacer planes o trabajo. Mitigar un riesgo podría requerir que el equipo construya algo para dejarlo a descubierto. Monitorear un riesgo implica elegir un mecanismo de monitoreo, instalarlo, y ejecutarlo. A su vez, mitigar o monitorear riesgos lleva esfuerzos de

desarrollo importantes, es decir, tiempo. Debido al tiempo que lleva tratar riesgos, una organización de un proyecto raramente puede tratar todos los riesgos al mismo tiempo. Esta es la razón por la que se hace necesaria la división por prioridades de las iteraciones. Esto es lo que se conoce como desarrollo iterativo conducido por los riesgos. Esta es una gestión de riesgos sensata.

### La Iteración Genérica

#### *¿Qué es una iteración?*

Una iteración es un miniproyecto –un viaje más o menos completo a través de todos los flujos de trabajo– que resulta en una versión interna. Esta es una noción intuitiva de lo que es una iteración.

En la Figura 3.41 se describen los elementos genéricos del flujo de trabajo de cada iteración. Todos pasan a través de los cinco flujos de trabajo centrales. Además, todos se inician con una actividad de planificación, y terminan con una evaluación.

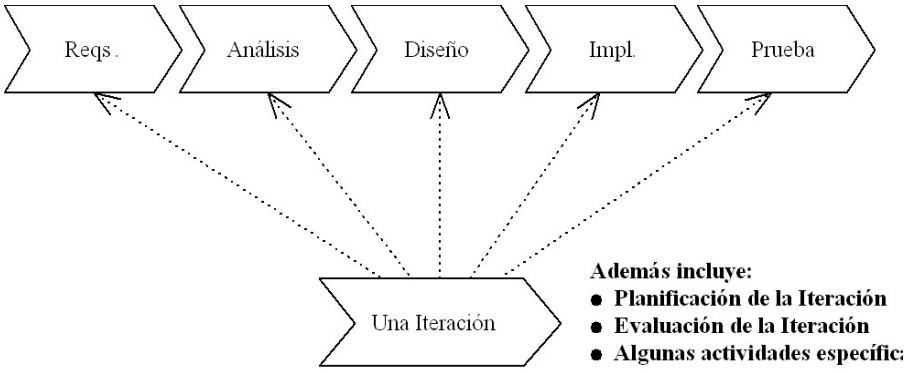


Figura 3.41: Una iteración atraviesa los cinco flujos de trabajo centrales

Cada flujo de trabajo central es una colaboración entre un conjunto de trabajadores y artefactos. Un *trabajador* representa una posición que puede asignarse a una persona o a un equipo, y especifica responsabilidades y habilidades requeridas. Ejemplos de trabajadores son el Analista de Sistemas, el Especificador de Casos de Uso, el Diseñador de Interfaces de Usuario, el Arquitecto, el Ingeniero de Componentes, etc. Un *artefacto* es cualquier tipo de descripción o información creada, producida, cambiada, o usada por los trabajadores al trabajar con el sistema. Ejemplos de artefactos son el Modelo de Casos de Uso, cada Caso de Uso en sí, un Actor, la Descripción de la Arquitectura, un Glosario, etc.

Existe una superposición entre iteraciones. Los trabajadores y artefactos pueden participar en más de un flujo de trabajo central. Por ejemplo, el ingeniero de componentes participa en tres flujos de trabajo: análisis, diseño, e implementación. Finalmente, el flujo de trabajo de la iteración se crea superponiendo un subconjunto elegido de los flujos de trabajo centrales unos sobre otros, y luego agregando lo demás, como la planificación y la evaluación.

Las primeras iteraciones se concentran en la comprensión del problema y de la tecnología. En la fase de inicio, las iteraciones se ocupan de producir un caso de negocio. En la fase de elaboración, las iteraciones están dirigidas al desarrollo de la arquitectura básica. En la fase de construcción, las iteraciones se concentran en darle forma al producto mediante una serie de construcciones dentro de cada iteración, terminando con un producto listo para ser entregado a la comunidad de usuarios. Sin embargo, cada iteración sigue el mismo patrón, como muestra la Figura 3.41.

Cada iteración se evalúa cuando termina. Un objetivo es determinar si han aparecido nuevos requerimientos, o si los existentes han cambiado, de forma que afecten a las iteraciones siguientes. En la planificación de la próxima iteración, el equipo también examina la forma en que los riesgos que aún existen afectarán el trabajo.

Las iteraciones pueden superponerse, en el sentido que una iteración puede comenzar antes de que la anterior haya sido completada, como muestra la Figura 3.42. Sin embargo, esta superposición no puede ir demasiado lejos, ya que una iteración siempre es la base para la siguiente.

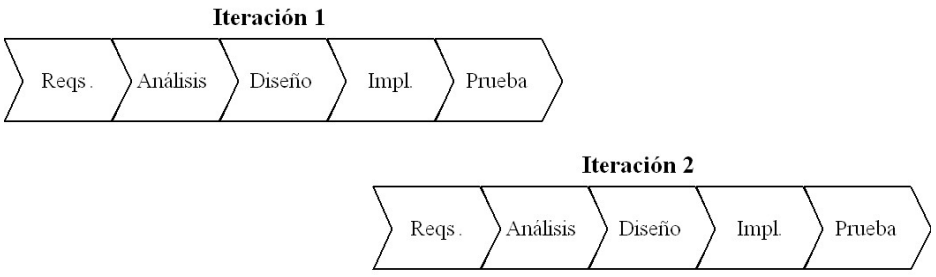


Figura 3.42: Iteraciones superpuestas

**El resultado de una iteración es un Incremento.**

Un *incremento* es la diferencia entre la versión interna de una iteración y la versión interna de la iteración siguiente. Al final de una iteración, el conjunto de modelos que

representa el sistema se encuentra en un *estado* particular. Cada modelo ha alcanzado un estado, y cada elemento esencial de un modelo está en un estado particular. Por ejemplo, el modelo de casos de uso al final de cada iteración contiene un conjunto de casos de uso que representa el grado en el cual la iteración ha llevado a cabo los requerimientos. Algunos de los casos de uso de este conjunto están completos, mientras que otros sólo están parcialmente completos. Al mismo tiempo, el modelo de diseño ha alcanzado un estado consistente con el modelo de casos de uso. Los subsistemas, interfaces, y realizaciones de casos de uso del modelo de diseño también están en estados consistentes unos con otros.

En cualquier punto de la secuencia de iteraciones, algunos subsistemas están completos. Contienen toda la funcionalidad prescripta, y ya han sido implementados y probados. Otros subsistemas sólo están parcialmente terminados, y otros aún están vacíos. Así, en términos más precisos, un incremento es la diferencia entre dos estados sucesivos.

### Iteraciones a lo largo del Ciclo de Vida

Cada una de las cuatro fases termina con un hito principal

- Inicio: objetivos del ciclo de vida.
- Elaboración: arquitectura del ciclo de vida.
- Construcción: capacidad operativa inicial.
- Transición: entrega del producto.

Los objetivos primarios de la fase de inicio son establecer el alcance de lo que el producto debería hacer, reducir los peores riesgos, y preparar el caso de negocio inicial, indicando que vale la pena proseguir con el proyecto desde el punto de vista del negocio. En otras palabras, se apunta a establecer los objetivos del ciclo de vida para el proyecto.

Los objetivos primarios de la fase de elaboración son delinear la arquitectura, capturar la mayoría de los requerimientos, y reducir los segundos peores riesgos, es decir, establecer la arquitectura del ciclo de vida. Al final de esta fase, se tiene la capacidad de estimar los costos y los tiempos y planificar la fase de construcción con cierto detalle.

Los objetivos primarios de la fase de construcción son desarrollar el sistema completo y asegurar que el producto puede comenzar su transición a los clientes, es decir, lograr su capacidad funcional inicial.

Los objetivos primarios de la fase de transición son asegurar que se tiene un producto listo para ser entregado a la comunidad de usuarios. Durante esta fase de desarrollo, los usuarios se entrenan en cómo usar el software.

Dentro de cada fase hay hitos menores, que son los criterios aplicables a cada iteración. Cada iteración produce resultados, *artefactos del modelo*. Así, al final de cada iteración, habrá un nuevo incremento a los modelos de casos de uso, de análisis, de diseño, de despliegue, de implementación, y de prueba. El nuevo incremento se integrará con los resultados de las iteraciones anteriores en una nueva versión del conjunto de modelos.

Si bien cada iteración es una pasada a través de los flujos de trabajo de requerimientos, análisis, diseño, implementación, y prueba, esas iteraciones ponen diferentes énfasis en las distintas fases. Durante las fases de inicio y elaboración, la mayor parte del esfuerzo está dirigida a la captura de los requerimientos y al análisis y diseño preliminar. Durante la construcción el énfasis se desplaza hacia el diseño, implementación, y prueba detallados. Las primeras fases tienen mucho de gestión del proyecto, y de desarrollo de un ambiente para el proyecto.

Los modelos también evolucionan con cada iteración. Cada una agrega un poco más de detalle a cada modelo. Algunos de estos modelos, como el de casos de uso, reciben más atención en las primeras etapas, mientras que otros, como el modelo de implementación, obtienen más atención durante la fase de construcción, como se muestra en la Figura 3.43. Las letras sobre cada barra representan un modelo diferente: U = Modelo de Casos de Uso; A = Modelo de Análisis; primera D = Modelo de Diseño; segunda D = Modelo de Despliegue; I = Modelo de Implementación; y P = Modelo de Pruebas. El trabajo en todos los modelos se continúa en todas las fases, como lo indica el llenado creciente de los modelos. La fase de construcción termina con un conjunto (casi) completo de modelos. Sin embargo, estos modelos necesitan de pequeños reajustes durante su transición, a medida que se distribuyen a la comunidad de usuarios.

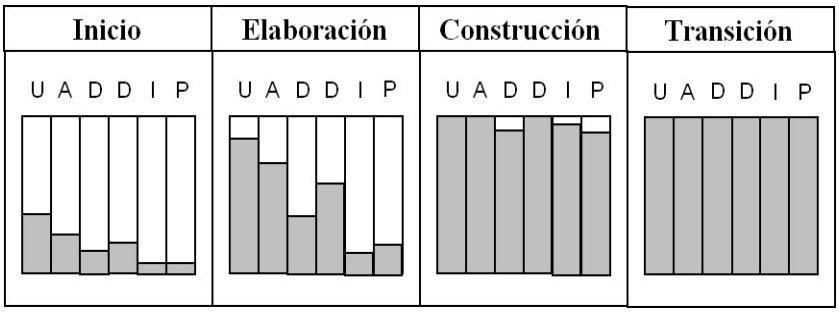


Figura 3.43: Evolución de los distintos modelos en cada etapa

## 3.3. El Lenguaje Unificado de Modelado (UML)

El UML es un lenguaje estándar para la escritura de diseños de software. El UML se puede usar para visualizar, especificar, construir, y documentar los artefactos de un sistema de software. Es un lenguaje muy expresivo, por lo que resulta adecuado para un rango muy amplio de sistemas. Pero la expresividad no implica la dificultad de comprensión y de uso. Aprender a usar el UML significa formarse un modelo conceptual del lenguaje, lo que requiere aprender tres elementos principales: los bloques de construcción básicos, las reglas que dictan cómo esos bloques se pueden poner juntos, y algunos mecanismos comunes que se aplican a través de todo el lenguaje.

El UML es sólo un lenguaje y por lo tanto es sólo una parte de un método de desarrollo de software. El UML es independiente del proceso, aunque idealmente debería ser usado en un proceso que sea manejado por casos de uso, centrado en la arquitectura, iterativo e incremental, como el Proceso Unificado descripto en al Sección 3.2.

### 3.3.1. Visión general del UML

El UML es un lenguaje para

- Visualizar
- Especificar
- Construir
- Documentar

los artefactos de un sistema de software.

### El UML es un *Lenguaje*

Un lenguaje provee un vocabulario y las reglas para combinar las palabras de ese vocabulario, con propósitos de comunicación. Un lenguaje de modelado es un lenguaje cuyo vocabulario y reglas se centran en las representaciones físicas y conceptuales de un sistema. Un lenguaje de modelado como el UML es entonces un lenguaje estándar para diseños de software.

El diseño de sistemas de software requiere de un lenguaje que trate las diferentes vistas de la arquitectura de un sistema a medida que evoluciona a lo largo del ciclo de vida de desarrollo del software. El vocabulario y las reglas de un lenguaje como el UML nos dicen cómo crear y leer modelos bien formados, pero no nos dicen qué modelos se deberían crear ni cuándo crearlos. Ese es el rol del proceso de desarrollo del software.

### El UML es un lenguaje para *Visualización*

Para muchos programadores, hay muy poca distancia entre pensar una implementación y codificarla. Se piensa, se codifica. Aún así, el programador realiza algo de modelado, quizás mental, quizás en un borrador. Sin embargo, este modo de trabajar acarrea varios problemas. Comunicar esos modelos conceptuales a otros es muy difícil, salvo que todos los involucrados utilicen el mismo lenguaje. Hay cosas relacionadas con sistemas de software que sólo pueden ser comprendidas luego de la construcción de modelos que trascienden el lenguaje de programación textual. Además, si el desarrollador que escribió el código nunca delineó los modelos que estaban en su cabeza, esa información se perdería para siempre si decidiera marcharse.

La escritura de modelos en UML soluciona los problemas anteriormente mencionados. La comunicación se realiza mucho más fácilmente utilizando modelos explícitos. Algunas cosas se modelan mejor textualmente, y otras se modelan mejor gráficamente. De hecho, en todos los sistemas interesantes hay estructuras que trascienden lo que se puede representar en un lenguaje de programación, y ahí se hacen notorias las bondades gráficas del UML. Detrás de cada símbolo de la notación UML hay una semántica bien definida. De esta manera, un desarrollador puede escribir un modelo en UML y otro desarrollador, o aún una herramienta, puede interpretar ese modelo sin ambigüedades.

### El UML es un lenguaje para *Especificación*

En este contexto, especificar significa construir modelos precisos, completos y sin ambigüedades. En particular, el UML trata la especificación de todas las decisiones importantes de análisis, diseño, e implementación que deben ser tomadas durante el desarrollo y la entrega de un sistema de software.

### El UML es un lenguaje para *Construcción*

El UML no es un lenguaje de programación visual, pero sus modelos pueden ser directamente conectados a una variedad de lenguajes de programación. Esto significa que es posible mapear un modelo en UML a un lenguaje de programación como Java, C++, o Visual Basic, o aún a tablas en una base de datos relacional o al almacenamiento permanente de una base de datos orientada a objetos.

Este mapeo permite la ingeniería hacia delante: la generación de código de un lenguaje de programación partiendo de un modelo UML. También es posible la inversa: se puede reconstruir un modelo UML a partir de una implementación. Esta ingeniería inversa no es mágica y requiere soporte, pero no es imposible.

Además de este mapeo directo, el UML es lo suficientemente expresivo e inequívoco como para permitir la ejecución directa de modelos, la simulación de sistemas, y la instrumentación de sistemas en operación.

### El UML es un lenguaje para *Documentación*

Una organización de software saludable produce toda clase de artefactos, además del código ejecutable propiamente dicho. Estos artefactos incluyen: requerimientos, arquitectura, diseño, código fuente, planes del proyecto, pruebas, prototipos, versiones, etc. Dependiendo de la cultura de desarrollo, algunos de estos artefactos son tratados más o menos formalmente que otros. Tales artefactos no son sólo los entregables de un proyecto, sino que también son críticos en el control, medición, y comunicación de un sistema durante su desarrollo y luego de su entrega.

El UML tiene en cuenta la documentación de la arquitectura de un sistema y todos sus detalles. También provee un lenguaje para expresar requerimientos y para las pruebas. Finalmente, el UML provee un lenguaje para modelar las actividades de planificación del proyecto y gestión de versiones.

#### ¿Dónde se puede usar el UML?

Se pretende que el UML sea usado primariamente para sistemas de software. Ha sido efectivamente usado en dominios tales como sistemas de información empresarial, servicios financieros y bancarios, telecomunicaciones, transporte, defensa, aeronáutica, comercio, medicina, ciencias, servicios basados en Web, etc. Sin embargo, el UML no se limita al modelamiento de software. De hecho, es lo suficientemente expresivo como para modelar sistemas que no sean de software, como flujos de trabajo en el sistema legal, la estructura y comportamiento de un sistema para pacientes cardíacos, o el diseño de hardware.

### 3.3.2. Un Modelo Conceptual del UML

Para comprender el UML es necesario formarse un modelo conceptual del lenguaje, y esto requiere el aprendizaje de tres elementos principales: los *bloques de construcción* básicos del UML, las *reglas* que dictan cómo se pueden reunir esos bloques de construcción, y algunos *mecanismos comunes* que se aplican a lo largo de todo el UML.

#### Bloques de Construcción del UML

El vocabulario del UML comprende tres tipos de bloques de construcción:

1. Cosas
2. Relaciones
3. Diagramas

Las cosas son las abstracciones más importantes en un modelo; las relaciones mantienen juntas estas cosas; los diagramas agrupan colecciones interesantes de cosas y sus relaciones.

##### *Cosas en el UML*

Hay cuatro tipos de cosas en el UML:

1. Cosas estructurales
2. Cosas de comportamiento

3. Cosas de agrupamiento

4. Cosas notacionales

Estas cosas son los bloques de construcción básicos orientados a objetos del UML. Se usan para escribir modelos bien formados.

**Cosas Estructurales.** Las *cosas estructurales* son los sustantivos de los modelos UML. En su mayoría son partes estáticas de un modelo, representando elementos que son conceptuales o físicos. En total, hay siete tipos de cosas estructurales, que se muestran en la Figura 3.44.

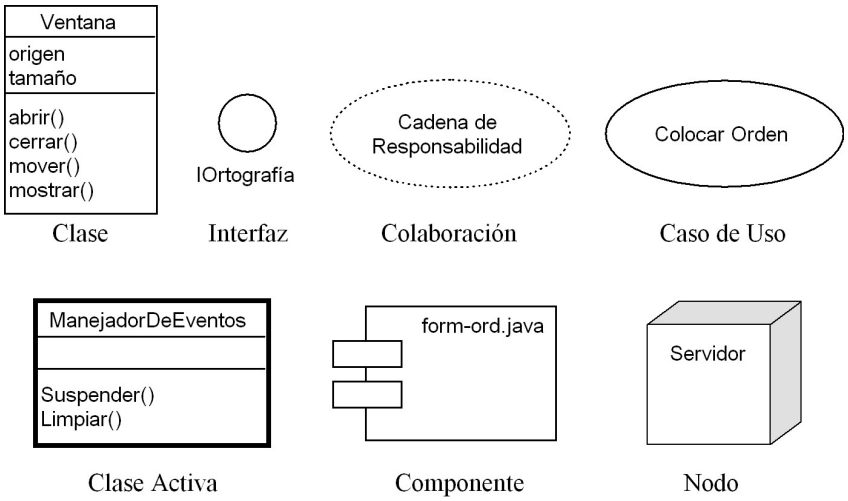


Figura 3.44: Cosas Estructurales en UML

Una *clase* es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. En UML, una clase se dibuja como un rectángulo, y usualmente incluye su nombre, atributos, y operaciones.

Una *interfaz* es una colección de operaciones que especifican un servicio de una clase o componente. Por lo tanto, describe el comportamiento externamente visible de ese elemento. En UML, una interfaz se representa como un círculo junto con su nombre, y raramente aparece sola. Lo más típico es que aparezca ligada a la clase o componente que la efectiviza.

Una *colaboración* define una interacción y es una sociedad de roles y otros elementos que trabajan juntos para proveer algún comportamiento cooperativo que es mayor a la suma de todos los elementos. En UML, una colaboración se dibuja como una elipse con líneas de puntos, y usualmente incluye sólo su nombre.

Un *caso de uso* es una descripción de un conjunto o secuencia de acciones que realiza un sistema y que logra un resultado observable de valor para un actor particular. En UML, un caso de uso se dibuja como una elipse con líneas sólidas, y usualmente incluye sólo su nombre.

Una *clase activa* es una clase cuyos objetos son dueños de uno o más procesos o hilos y por lo tanto pueden iniciar una actividad de control. En UML, una clase activa se representa igual que una clase, pero con bordes gruesos. También generalmente incluye su nombre, atributos, y operaciones.

Un *componente* es una parte física y sustituible de un sistema que conforma y provee la realización de un conjunto de interfaces. En un sistema se encontrarán diferentes tipos de componentes de desarrollo, como componentes COM+ o Java Beans, y además componentes que son artefactos del proceso de desarrollo, como archivos de código fuente. En UML, un componente se dibuja como un rectángulo con lengüetas, y usualmente incluye sólo su nombre.

Por último, un *nodo* es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional, que generalmente tiene al menos algo de memoria y con frecuencia, capacidad de procesamiento. Un conjunto de componentes puede residir en un nodo y también puede migrar de un nodo a otro. En UML, un nodo se representa con un cubo, y usualmente sólo incluye su nombre.

Estos siete elementos son las cosas estructurales básicas que se pueden incluir en un modelo UML. También existen variaciones de estos, como actores, señales, y utilidades (tipos de clases), procesos e hilos (tipos de clases activas), y aplicaciones, documentos, archivos, librerías, páginas, y tablas (tipos de componentes).

**Cosas de Comportamiento.** Las *cosas de comportamiento* son las partes dinámicas de los modelos UML. Son los verbos de un modelo, y representan comportamiento a través del tiempo y del espacio. Existen dos tipos primarios de cosas de comportamiento, cuyos principales elementos se muestran en la Figura 3.45.

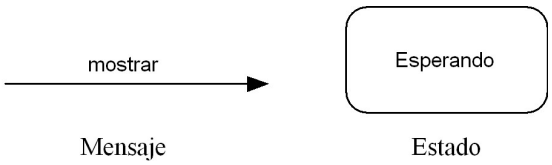


Figura 3.45: Cosas de Comportamiento en UML

Una *interacción* es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular y para cumplir con un propósito específico. El comportamiento de una sociedad de objetos o de una operación individual se pueden especificar con una interacción. Una interacción comprende un número de otros elementos, incluyendo mensajes, secuencias de acción (el comportamiento activado por un mensaje), y enlaces (la conexión entre objetos). Gráficamente, un mensaje se representa como una flecha, casi siempre con el nombre de su operación (Figura 3.45).

Una *máquina de estados* es un comportamiento que especifica la secuencia de estados por los que pasa un objeto durante su vida en respuesta a eventos, junto con sus respuestas a esos eventos. Con una máquina de estados se puede especificar el comportamiento de una clase individual o de una colaboración de clases. Una máquina de estados comprende un número de otros elementos, incluyendo estados, transiciones (el flujo de un estado a otro), eventos (cosas que disparan una transición), y actividades (la respuesta a una transición). Gráficamente, un estado se dibuja como un rectángulo redondeado, y usualmente incluye su nombre y subestados, si hubiere (Figura 3.45).

Estos dos elementos –interacciones y máquinas de estados– son las cosas de comportamiento básicas que se pueden incluir en un modelo UML. Semánticamente, estos elementos están por lo general conectados a varios elementos estructurales, principalmente clases, colaboraciones y objetos.

**Cosas de agrupamiento.** Las *cosas de agrupamiento* son las partes organizacionales de los modelos UML. Son las cajas dentro de las cuales se puede descomponer un modelo. Hay un solo tipo de cosa de agrupamiento, los paquetes.

Un *paquete* es un mecanismo de propósito general para organizar elementos en grupos. Las cosas estructurales, las cosas de comportamiento, e incluso otras cosas de agrupamiento se pueden colocar en un paquete. A diferencia de los componentes (que existen en tiempo de ejecución), un paquete es puramente conceptual (es decir que existe sólo en tiempo de desarrollo). Gráficamente, un paquete se representa como una carpeta con lengüeta, usualmente incluyendo su nombre, y a veces su contenido, como muestra la Figura 3.46.

Los paquetes son las cosas de agrupamiento básicas con las cuales se puede organizar un modelo UML. También hay variaciones, tales como marcos de trabajo,

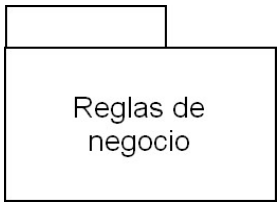


Figura 3.46: Paquetes

modelos, y subsistemas (tipos de paquetes).

**Cosas notacionales.** Las *cosas notacionales* son las partes explicativas de los modelos UML. Son los comentarios que se pueden aplicar para describir, aclarar, y remarcar cualquier elemento de un modelo. Hay una clase principal de cosa notacional, llamada *nota*. Una nota es simplemente un símbolo para representar restricciones y comentarios adjuntos a un elemento o una colección de elementos. Gráficamente, una nota se dibuja como un rectángulo con una esquina doblada, junto con un comentario textual o gráfico, como en la Figura 3.47.

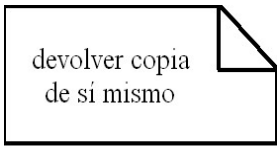


Figura 3.47: Notas

Este elemento es la única cosa notacional que se puede incluir en un modelo UML. Típicamente se usarán notas para adornar los diagramas con restricciones y comentarios que se expresan mejor en texto formal o informal. También hay variaciones de estos elementos, tales como requerimientos (que especifican algún comportamiento deseado desde una perspectiva exterior al modelo).

***Relaciones en el UML***

Hay cuatro tipos de relaciones en el UML:

1. Dependencia
2. Asociación
3. Generalización

4. Realización

Estas relaciones son los bloques de construcción relacional básicos del UML, y se muestran en la Figura 3.48. Se usan para escribir modelos bien formados.

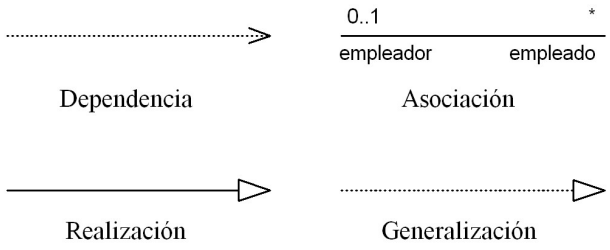


Figura 3.48: Relaciones en UML

Una *dependencia* es una relación semántica entre dos cosas en la cual un cambio a una cosa (la cosa independiente) puede afectar la semántica de la otra cosa (la cosa dependiente). En UML, una dependencia se representa como una línea de puntos, posiblemente dirigida, e incluye ocasionalmente una etiqueta.

Una *asociación* es una relación estructural que describe un conjunto de enlaces o conexiones entre objetos. En UML, una asociación se dibuja como una línea sólida, posiblemente dirigida, ocasionalmente incluye una etiqueta, y con frecuencia contiene otros adornos, como multiplicidad y nombres de roles. En el ejemplo de asociación de la Figura 3.48, *empleador* y *empleado* son los nombres de los *roles* que desempeñan las clases unidas por la asociación, mientras que la *multiplicidad* indica cuántos objetos pueden conectarse por medio de una instancia de la asociación. En este caso, un *empleador* puede tener varios *empleados* (*), pero un *empleado* puede no tener *empleador* o tener sólo uno (0..1).

Una *generalización* es una relación de herencia, en la cual los objetos del elemento especializado (el hijo) son sustituibles por objetos del elemento generalizado (el padre). De esta forma, el hijo comparte la estructura y el comportamiento del padre. En UML, una relación de generalización se representa como una flecha sólida con la cabeza hueca apuntando al padre.

Una *realización* es una relación semántica entre clasificadores, donde un clasificador especifica un contrato que otro clasificador se compromete a cumplir. Se encontrarán relaciones de realización en dos lugares: entre interfaces y las clases o componentes que las realizan, y entre casos de uso y las colaboraciones que los realizan.

Gráficamente, una relación de realización se representa como una cruza entre una generalización y una relación de dependencia.

Estos cuatro elementos son las cosas relacionales básicas que se pueden incluir en un modelo UML. También hay variaciones, como refinamientos, traza, incluye, y extiende (para dependencias).

***Diagramas en el UML***

Un *diagrama* es la presentación gráfica de un conjunto de elementos, más frecuentemente representado como un grafo conectado de vértices (cosas) y arcos (relaciones). Se dibujan diagramas para visualizar un sistema desde diferentes perspectivas, por lo tanto un diagrama es una proyección de un sistema. Un diagrama representa una vista abreviada de los elementos que constituyen un sistema. El mismo elemento puede aparecer en todos los diagramas, en unos pocos diagramas (el caso más común), o en ninguno. En teoría, un diagrama puede contener cualquier combinación de cosas y relaciones. Sin embargo, en la práctica surge un pequeño número de combinaciones comunes, las cuales son consistentes con las cinco visiones más útiles que comprenden la arquitectura de un sistema de software. Por esta razón, el UML incluye nueve de tales diagramas:

1. Diagrama de clases.
2. Diagrama de objetos.
3. Diagrama de casos de uso.
4. Diagrama de secuencia.
5. Diagrama de colaboración.
6. Diagrama de estados.
7. Diagrama de actividad.
8. Diagrama de componentes.
9. Diagrama de despliegue.

Esta no es una lista cerrada de diagramas. Ciertas herramientas pueden usar el UML para proveer otros tipos de diagramas, aunque estos nueve son por lejos los más comunes que se pueden encontrar en la práctica.

Un *diagrama de clases* muestra un conjunto de clases, interfaces, y colaboraciones y sus relaciones. Los diagramas de clases son los diagramas más comunes que se pueden encontrar al modelar sistemas orientados a objetos.

Los diagramas de clases se utilizan para modelar la vista de diseño estática de un sistema. En su mayor parte, esto implica modelar el vocabulario del sistema, las colaboraciones, o los esquemas. Los diagramas de clases son también la base para un par de diagramas relacionados: los diagramas de componentes y los diagramas de distribución.

Por ejemplo, la Figura 3.49 muestra un conjunto de clases surgidas de la implementación de un robot autónomo. La figura se enfoca en las clases implicadas en el mecanismo para mover el robot a lo largo de un camino. Puede verse una clase abstracta (*Motor*) con dos descendientes concretos, *MotorDeDirección* y *MotorPrincipal*. Ambas clases heredan las cinco operaciones de su padre, *Motor*. Las dos clases se muestran, a su vez, como partes de otra clase, *Conductor*. La clase *AgenteCamino* tiene una asociación uno-a-uno con *Conductor* y una asociación uno-a-muchos con *SensorDeColisiones*. No se muestran atributos ni operaciones para *AgenteCamino*, aunque sí se dan sus responsabilidades.

Hay muchas más clases implicadas en este sistema, pero este diagrama se enfoca sólo en aquellas abstracciones que están directamente involucradas en mover el robot.

Un *diagrama de objetos* muestra un conjunto de objetos y sus relaciones en un momento dado. Los diagramas de objetos modelan las instancias de las cosas contenidas en los diagramas de clases.

Los diagramas de objetos se usan para modelar la vista estática de diseño o de procesos de un sistema. Esto implica modelar una instantánea del sistema en un momento preciso del tiempo y representar un conjunto de objetos, su estado, y sus relaciones.

Por ejemplo, la Figura 3.50 muestra un conjunto de objetos surgidos de la implementación de un robot autónomo, que respeta el diagrama de clases de la Figura 3.49. Esta figura se enfoca en algunos de los objetos involucrados en el mecanismo usado por el robot para calcular un modelo del mundo en el cual se mueve. Hay muchos

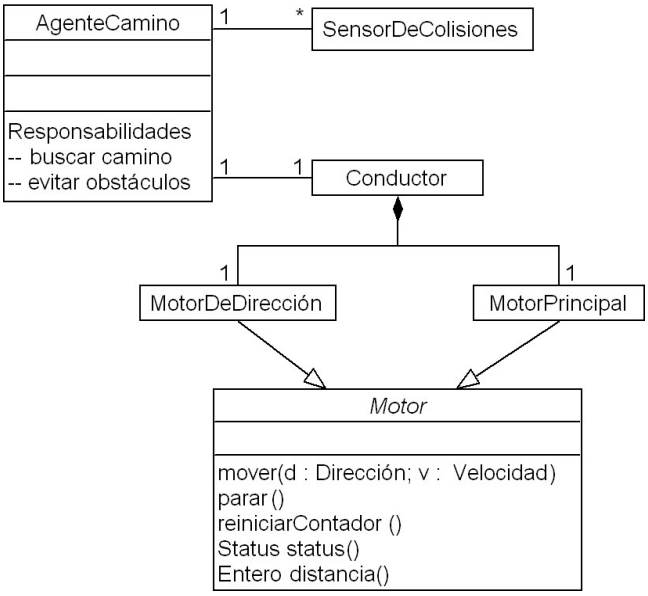


Figura 3.49: Un Diagrama de Clases

otros objetos involucrados en un sistema en ejecución, pero este diagrama se enfoca sólo en aquellas abstracciones que están directamente implicadas en crear esta visión del mundo.

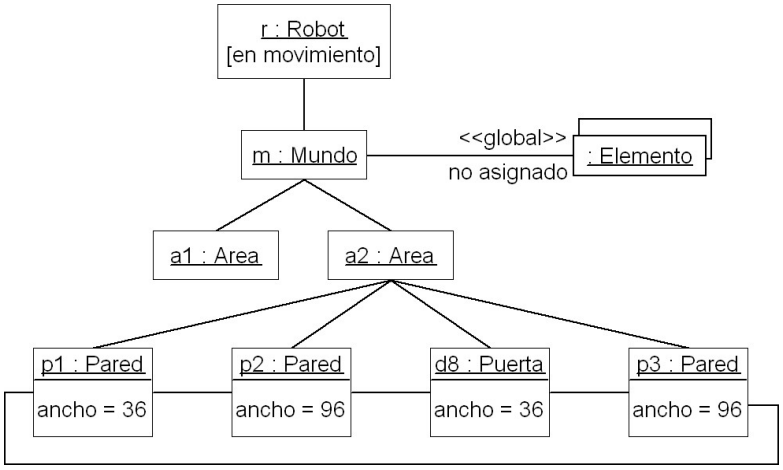


Figura 3.50: Un Diagrama de Objetos

Como indica esta figura, un objeto representa el robot mismo ($r$, una instancia de *Robot*), y $r$ está actualmente en el estado marcado como *en movimiento*. Este objeto tiene un vínculo a $m$, una instancia de *Mundo*, que representa una abstracción del

modelo de mundo del robot. Este objeto tiene un enlace a un multiobjeto que consiste de instancias de *Elemento*, que representan entidades que el robot ha identificado pero que todavía no ha asignado en su visión del mundo. Estos elementos están marcados como parte del estado global del robot.

En este momento del tiempo, *m* está enlazado a dos instancias de *Area*. Una de ellas (*a2*) se muestra con sus propios enlaces a tres objetos *Pared* y un objeto *Puerta*. Cada una de estas paredes está marcada con su ancho actual, y cada una se muestra vinculada a sus paredes adyacentes. Como sugiere este diagrama de objetos, el robot ha reconocido esta área encerrada, que tiene paredes en tres lados y una puerta en el cuarto lado.

Un *diagrama de casos de uso* muestra un conjunto de casos de uso y actores (un tipo especial de clase) y sus relaciones. Los diagramas de casos de uso sirven para modelar los aspectos dinámicos de los sistemas, es decir el comportamiento de un sistema, un subsistema, o una clase. Con ellos se puede visualizar, especificar, y documentar el comportamiento de un elemento.

Por ejemplo, la Figura 3.51 muestra el contexto de un sistema de validación de tarjetas de crédito, con un énfasis en los actores que rodean el sistema. Existen *Clientes*, de los cuales hay dos tipos (*Cliente individual* y *Cliente empresarial*). Estos actores son los roles que los humanos desempeñan cuando interactúan con el sistema. En este contexto, también hay actores que representan otras instituciones, como *Institución comercial* (con la cual un *Cliente* realiza una transacción de tarjeta para comprar un producto o servicio) e *Institución financiera patrocinante* (que sirve como la banca de liquidación para la cuenta de la tarjeta de crédito). En el mundo real, estos últimos dos actores probablemente son en sí mismos sistemas de software.

Los diagramas de secuencia y de colaboración –ambos llamados *diagramas de interacción*– también sirven para modelar los aspectos dinámicos de los sistemas. Un diagrama de interacción muestra un conjunto de objetos y sus relaciones, incluyendo los mensajes que pueden ser enviados entre ellos. Un *diagrama de secuencia* es un diagrama de interacción que enfatiza el ordenamiento en el tiempo de los mensajes; un *diagrama de colaboración* es un diagrama de interacción que enfatiza la organización estructural de los objetos que envían y reciben mensajes. Los diagramas de secuencia y de colaboración son *isomorfos*, lo que significa que se puede tomar uno y transformarlo en el otro.

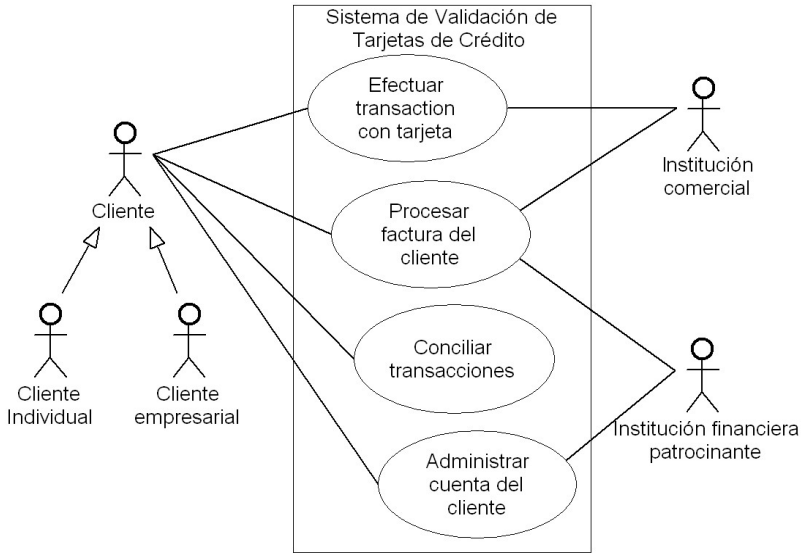


Figura 3.51: Un Diagrama de Casos de Uso

Como muestra la Figura 3.52, los diagramas de secuencia se forman ubicando primero los objetos que participan en la interacción en la parte superior del diagrama, a lo largo del eje X. Típicamente, los objetos que inician la interacción se ubican a la izquierda, y en forma creciente hacia la derecha los objetos más subordinados. Luego se colocan los mensajes que estos objetos envían y reciben a lo largo del eje Y en orden de tiempo creciente de arriba hacia abajo. Esto le da al lector una pista visual clara del flujo de control en el tiempo.

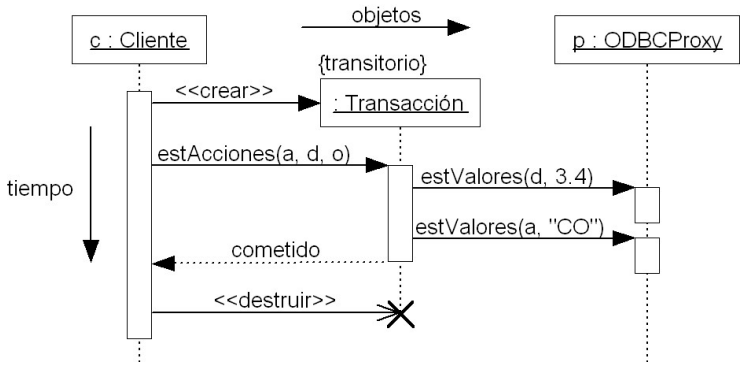


Figura 3.52: Un Diagrama de Secuencia

Los diagramas de secuencia tienen dos características que los distinguen de los diagramas de colaboración. Primero, está la *línea de vida* del objeto. Una línea de

vida de un objeto es la línea punteada vertical que representa la existencia de un objeto en un período de tiempo. La mayoría de los objetos que aparecen en un diagrama de interacción existirán mientras dure la interacción, por lo que estos objetos están todos alineados en la parte superior del diagrama, con sus líneas de vida dibujadas desde la parte superior del diagrama hasta la parte inferior. Los objetos pueden ser creados durante la interacción. Sus líneas de vida comienzan con la recepción del mensaje estereotipado *crear*. Los objetos también pueden destruirse durante la interacción. Sus líneas de vida finalizan con la recepción del mensaje estereotipado *destruir* (y se le agrega una gran X, marcando el fin de su vida).

Segundo, está el *foco de control*. El foco de control es un rectángulo alto y delgado que muestra el período de tiempo durante el cual un objeto está ejecutando una acción, ya sea directamente o mediante un procedimiento subordinado. La parte superior del rectángulo se alínea con el comienzo de la acción; la parte inferior se alínea con su finalización (y puede marcarse mediante un mensaje de retorno).

Un diagrama de colaboración enfatiza la organización de los objetos que participan en una interacción. Como muestra la Figura 3.53, un diagrama de colaboración se construye colocando primero los objetos que participan en la interacción como los vértices de un grafo. Luego, los enlaces que conectan estos objetos se representan como los arcos de este grafo. Por último, estos enlaces se adornan con los mensajes que los objetos envían y reciben. Esto le da al lector una pista visual clara del flujo de control en el contexto de la organización estructural de los objetos que colaboran.

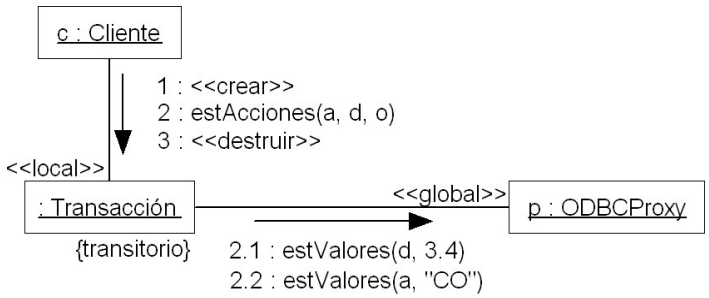


Figura 3.53: Un Diagrama de Colaboración

Los diagramas de colaboración tienen dos características que los distinguen de los diagramas de secuencia. Primero, está el *camino*. Para indicar cómo se conecta un objeto con otro, se puede adjuntar un estereotipo de camino al extremo más lejano

de un enlace (como «*local*», indicando que el objeto especificado es local al emisor).

Segundo, está el *número de secuencia*. Para indicar el orden de un mensaje en el tiempo, se precede el mensaje con un número (comenzando con el mensaje numerado como *1*), aumentando monótonamente para cada nuevo mensaje del flujo de control (*2*, *3*, etc.). Para mostrar anidamiento, se usa la numeración decimal de Dewey (*1.1*, *1.2*, etc. A lo largo del mismo enlace, se pueden mostrar varios mensajes (posiblemente enviados desde diferentes direcciones), y cada uno tendrá un único número de secuencia.

Un *diagrama de estados* muestra una máquina de estados, consistente de estados, transiciones, eventos, y actividades. Los diagramas de estados tratan la vista dinámica de un sistema. Son especialmente importantes para modelar el comportamiento de una interfaz, clase, o colaboración y enfatiza el comportamiento basado en eventos de un objeto, lo que es especialmente útil para modelar sistemas reactivos.

Por ejemplo, la Figura 3.54 muestra el diagrama de estados para analizar gramáticamente un lenguaje simple libre de contexto, como el que podríamos encontrar en sistemas que traduzcan mensajes desde o hacia XML. En este caso, la máquina está diseñada para analizar una cadena de caracteres que se corresponde con la sintaxis

$$message : \text{'<'}\ string\ \text{'>'}\ string\ \text{';'}$$

El primer string representa una etiqueta; el segundo string representa el cuerpo del mensaje. Dada una cadena de caracteres, sólo se pueden aceptar los mensajes bien formados que siguen esta sintaxis.

Como muestra la Figura 3.54, hay sólo tres estados estables para esta máquina de estados: *Esperando*, *Obteniendo Señal*, y *Obteniendo Cuerpo*. Mientras está en el estado *Esperando*, la máquina desecha cualquier caracter que no represente el comienzo de una secuencia válida (como se especifica en la condición guarda). Cuando se recibe el comienzo de una secuencia válida, el estado del objeto cambia a *Obteniendo Señal*. Mientras está en ese estado, la máquina almacena todos los caracteres que no designan el final de una secuencia válida (como se especifica en la condición guarda). Cuando se recibe el final de una secuencia válida, el estado del objeto cambia a *Obteniendo Cuerpo*. Mientras está en ese estado, la máquina almacena todos los caracteres que no designan el final del cuerpo de un mensaje (como se especifica en la condición guarda). Cuando se recibe el final de un mensaje, el estado del objeto cam-

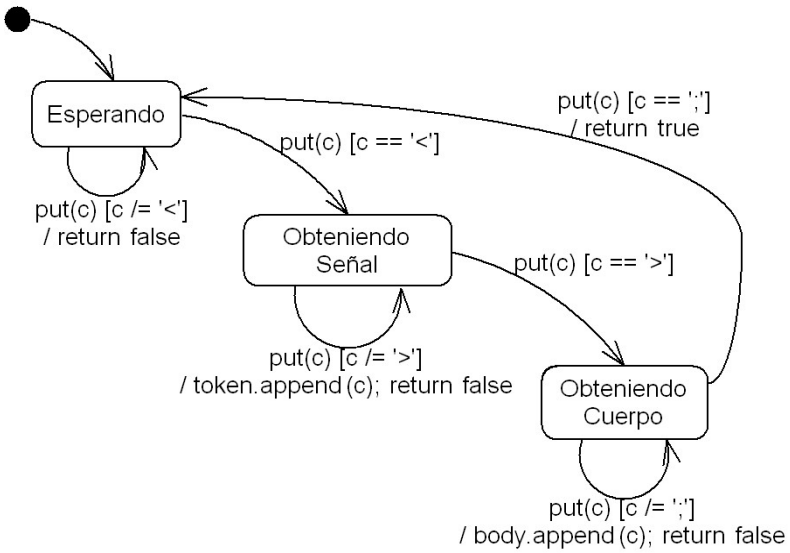


Figura 3.54: Un Diagrama de Estados

bia a *Esperando*, y se devuelve un valor que indica que el mensaje ha sido analizado (y la máquina está lista para recibir otro mensaje).

Un *diagrama de actividad* es un tipo especial de diagrama de estados que muestra el flujo de actividad a actividad dentro de un sistema. Estos diagramas también representan la vista dinámica de un sistema, y son especialmente importantes para modelar la función de un sistema y enfatizar el flujo de control entre objetos.

Mientras que los diagramas de interacción enfatizan el flujo de control de objeto en objeto, los diagramas de actividad enfatizan el flujo de control de actividad en actividad. Una actividad es una ejecución no atómica en curso dentro de una máquina de estados. Las actividades, al final, resultan en alguna acción, formada por computaciones atómicas ejecutables que resultan en un cambio de estado del sistema o el retorno de un valor.

Por ejemplo, la Figura 3.55 muestra un diagrama de actividad para un comercio, que especifica el flujo de trabajo implicado cuando un cliente devuelve un ítem de una orden de correo. El trabajo comienza con la acción *Solicita devolución* de *Cliente* y luego fluye a través de *Televentas* (*Obtiene número de devolución*), de nuevo al *Cliente* (*Envía Ítem*), luego a *Depósito* (*Recibe Ítem* y *Repone Ítem*), y finalizando en *Contabilidad* (*Abona cuenta*). Como indica el diagrama, también hay un objeto significativo en el flujo del proceso (*i*, una instancia de *Ítem*), que pasa del estado

*devuelto* al de *disponible*.

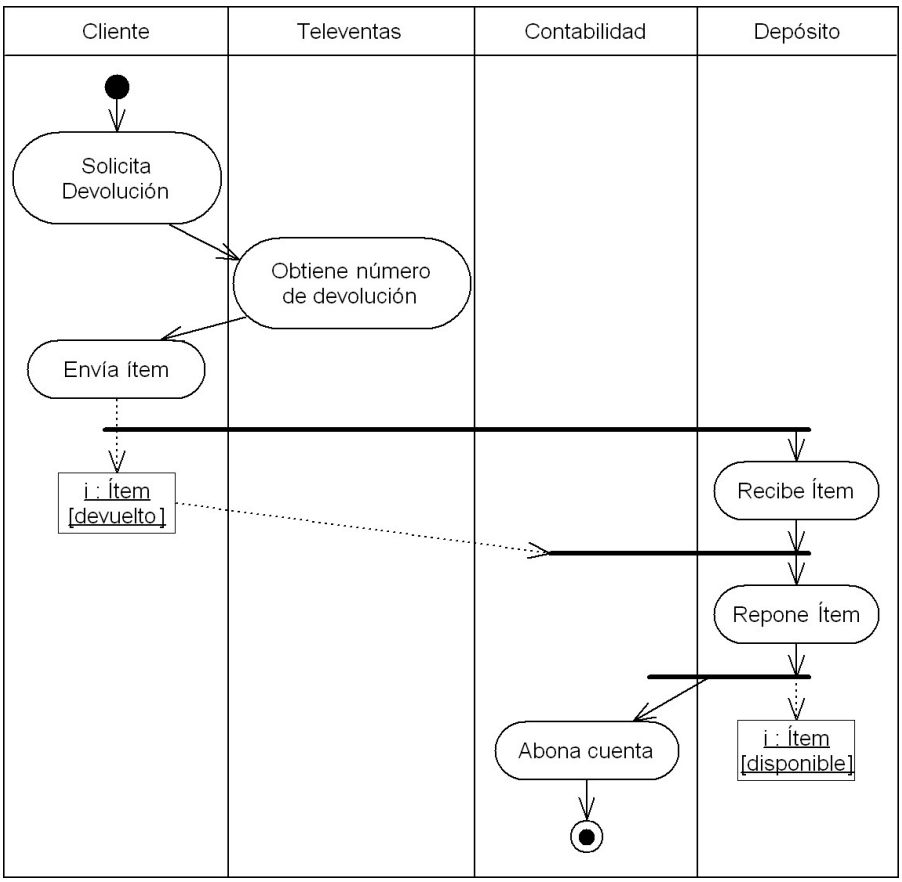


Figura 3.55: Un Diagrama de Actividad

Un *diagrama de componentes* muestra la organización y dependencias entre un conjunto de componentes. Muestra la vista de implementación estática de un sistema. Los diagramas de componentes están relacionados con los diagramas de clases, ya que un componente típicamente mapea a una o más clases, interfaces, o colaboraciones.

Los diagramas de componentes se usan para modelar las cosas físicas que residen en un nodo, como ejecutables, bibliotecas, tablas, archivos, y documentos. Los diagramas de componentes son esencialmente diagramas de clases que se concentran en los componentes de un sistema.

Por ejemplo, la Figura 3.56 modela parte de la edición ejecutable para un robot autónomo. Esta figura se concentra en los componentes de distribución asociados con las funciones de manejo y cálculo del robot. Existe un componente (*conductor.dll*) que exporta una interfaz (*IConducir*) que es, a su vez, importada por otro componente

(*camino.dll*). El componente *conductor.dll* exporta otra interfaz (*IProbarse*) que probablemente es usada por otros componentes del sistema, aunque aquí no se muestran. En el diagrama se muestra otro componente (*colisión.dll*) que también exporta un conjunto de interfaces, aunque se han omitido estos detalles: *camino.dll* se muestra con una dependencia directamente hacia *colisión.dll*.

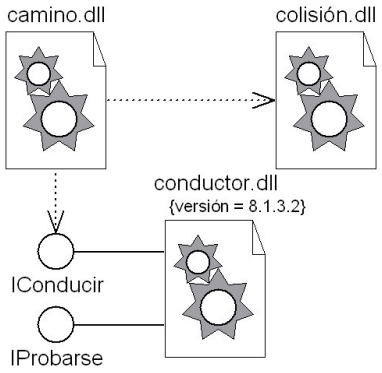


Figura 3.56: Un Diagrama de Componentes que modela una Versión Ejecutable

La Figura 3.57 muestra un conjunto de tablas de base de datos surgidas de un sistema de información para una escuela. Existe una base de datos (*escuela.db*, representada por un componente estereotipado como *base de datos*) que está compuesta por cinco tablas: *curso*, *departamento*, *instructor*, *clase*, y *estudiante*.

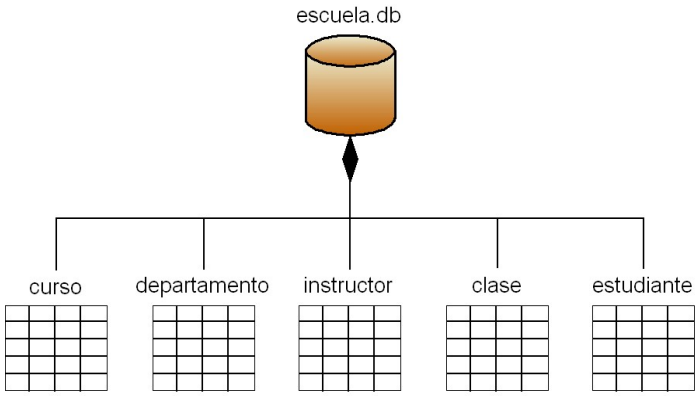


Figura 3.57: Un Diagrama de Componentes que modela una Base de Datos Física

Un *diagrama de despliegue* muestra la configuración de los nodos de procesamiento en tiempo de ejecución y los componentes que residen en ellos. Se usan para modelar

los aspectos físicos de un sistema orientado a objetos, y muestran la vista de despliegue estática de una arquitectura. Están relacionados con los diagramas de componentes, ya que un nodo encierra típicamente uno o más componentes.

La Figura 3.58 muestra la topología de un sistema completamente distribuido. Este diagrama de despliegue particular es también un diagrama de objetos, ya que contiene sólo instancias. Podemos ver tres consolas (instancias anónimas del nodo estereotipado *consola*), que están enlazados a la *Internet* (claramente un nodo único). En este diagrama, la Internet ha sido representada como un nodo estereotipado. A su vez, hay tres instancias de *servidor regional*, que sirven como interfaces de los *servidores nacionales*, de los cuales se muestra sólo uno. Como indica la nota, los servidores nacionales están conectados entre sí, pero sus relaciones no se muestran en este diagrama.

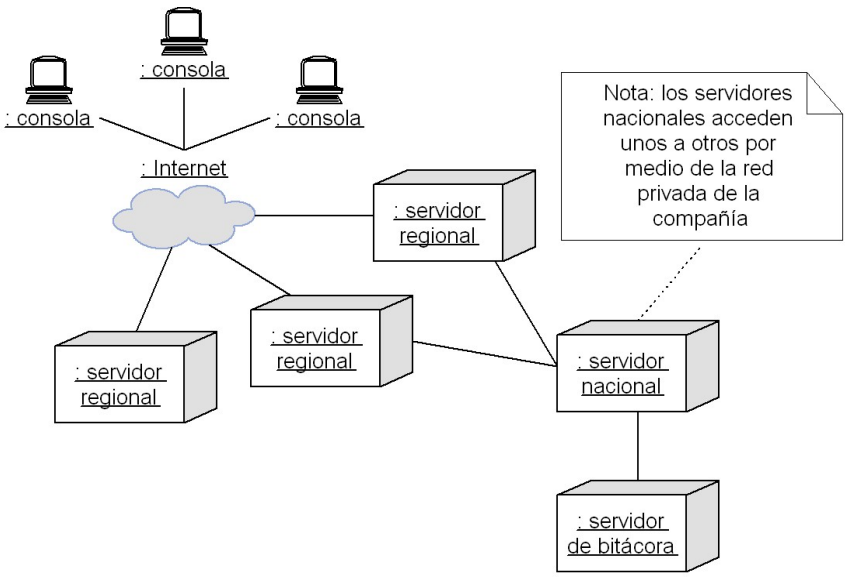


Figura 3.58: Un Diagrama de Despliegue

### Reglas del UML

Como cualquier lenguaje, el UML tiene un número de reglas que especifican cómo debería ser un modelo bien formado. Un *modelo bien formado* es un modelo que es semánticamente consistente y en armonía con todos los modelos relacionados.

El UML tiene reglas semánticas para

- Nombres: a qué se le llama cosas, relaciones, y diagramas.
- Alcance: el contexto que da un significado específico a un nombre.
- Visibilidad: cómo esos nombre pueden ser vistos y usados por otros.
- Integridad: cómo se relacionan apropiada y consistentemente unas cosas con otras.
- Ejecución: qué significa correr o simular un modelo dinámico.

**Mecanismos comunes en el UML**

El UML está simplificado por la presencia de cuatro mecanismos comunes que se aplican a lo largo de todo el lenguaje.

1. Especificaciones
2. Adornos
3. Divisiones comunes
4. Mecanismos de extensibilidad

***Especificaciones***

El UML es más que un mero lenguaje gráfico. Detrás de cada parte de su notación gráfica hay una especificación que provee una declaración textual de la sintaxis y semántica de ese bloque de construcción. Se usa la notación gráfica del UML para visualizar un sistema; se usa la especificación del UML para establecer los detalles del sistema. Dada esta separación, es posible construir un modelo en forma incremental, dibujando diagramas y luego agregando semántica a las especificaciones del modelo, o directamente creando una especificación y luego creando diagramas que sean proyecciones de esas especificaciones.

Las especificaciones del UML proveen una semántica de segundo plano que contiene todas las partes de todos los modelos de un sistema, y donde cada parte está relacionada con las demás en forma consistente. Los diagramas UML son por lo tanto simples proyecciones visuales de ese segundo plano, y cada diagrama revela un aspecto de interés específico del sistema.

***Adornos***

La mayoría de los elementos en el UML tienen una notación gráfica única y directa que provee una representación visual de los aspectos más importantes del elemento. Por ejemplo, la notación de una clase está intencionalmente diseñada para ser fácil de dibujar, ya que las clases son los elementos más comunes en el modelamiento de sistemas orientados a objetos. La notación de clase también expone los aspectos más importantes de una clase, como su nombre, atributos, y operaciones.

La especificación de una clase puede incluir otros detalles, como si es abstracta o la visibilidad de sus atributos y operaciones. Muchos de estos detalles se pueden representar como adornos textuales o gráficos en la notación rectangular básica de una clase. Por ejemplo, la Figura 3.59 muestra una clase adornada para indicar que es una clase abstracta (nombre en cursiva) con dos operaciones públicas (+), una protegida (#), y una privada (-).

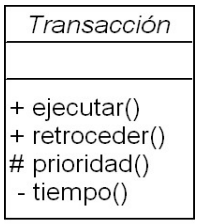


Figura 3.59: Adornos

Cada elemento en la notación UML comienza con un símbolo básico al cual se le pueden agregar una variedad de adornos específicos de ese símbolo.

***Divisiones comunes***

En el modelado de sistemas orientados a objetos, el mundo con frecuencia se divide en por lo menos un par de formas. Primero, la división de clases y objetos. Una clase es una abstracción; un objeto es una manifestación concreta de esa abstracción. En UML se pueden modelar tanto las clases como los objetos, como muestra la Figura 3.60.

En esta figura hay una clase, llamada *Cliente*, junto con tres objetos: *Juan* (que está marcado explícitamente como perteneciente a la clase *Cliente*), *:Cliente* (un objeto *Cliente* anónimo), y *Elisa* (que en su especificación está marcada como un objeto de clase *Cliente*, aunque no se muestra explícitamente aquí).

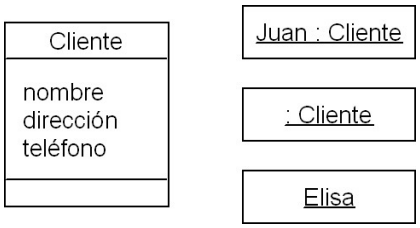


Figura 3.60: Clases y Objetos

Casi todos los bloques de construcción en UML tienen este mismo tipo de dicotomía clase/objeto. Gráficamente, el UML distingue un objeto usando el mismo símbolo que su clase y simplemente subrayando el nombre del objeto.

Segundo, hay una separación entre interfaz e implementación. Una interfaz declara un contrato, y una implementación representa una realización concreta de ese contrato, responsable de llevar a cabo fielmente la semántica completa de la interfaz. En UML se puede modelar tanto las interfaces como sus implementaciones, como se muestra en la Figura 3.61.

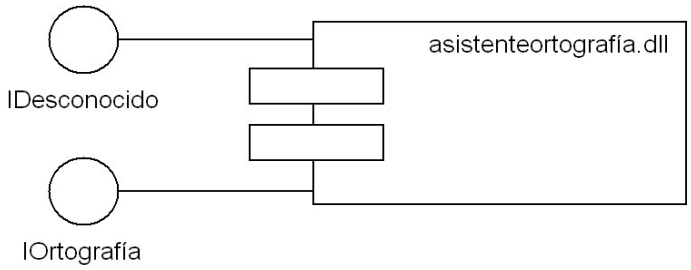


Figura 3.61: Interfaces e Implementaciones

En esta figura hay un componente llamado *asistenteortografía.dll* que implementa dos interfaces, *IDesconocido* e *IOrtografía*. Casi todos los bloques de construcción en UML tienen este mismo tipo de dicotomía interfaz/implementación.

***Mecanismos de extensibilidad***

El UML provee un lenguaje estándar para escribir diseños de software, pero no es posible para un lenguaje cerrado abarcar todos los matices de todos los modelos, en todos los dominios y en todo tiempo. Es por esto que el UML es abierto, haciendo posible la extensión del lenguaje en forma controlada. Los mecanismos de extensibilidad del UML incluyen

- Estereotipos
- Valores etiquetados
- Restricciones

Un *estereotipo* extiende el vocabulario del UML, permitiendo la creación de nuevos tipos de bloques de construcción derivados de los existentes pero específicos para cada problema. Por ejemplo, con frecuencia se podría querer modelar excepciones. En lenguajes como Java o C++ las excepciones no son otra cosa que clases, aunque sean tratadas de forma muy especial. Por lo general, sólo se desea que una excepción sea disparada o capturada, nada más. Se puede hacer que las excepciones sean "ciudadanos de primera clase" en el modelo –es decir, que sean tratadas como bloques de construcción básicos– marcándolas con un estereotipo apropiado, como en la clase *desbordamiento* de la Figura 3.62.

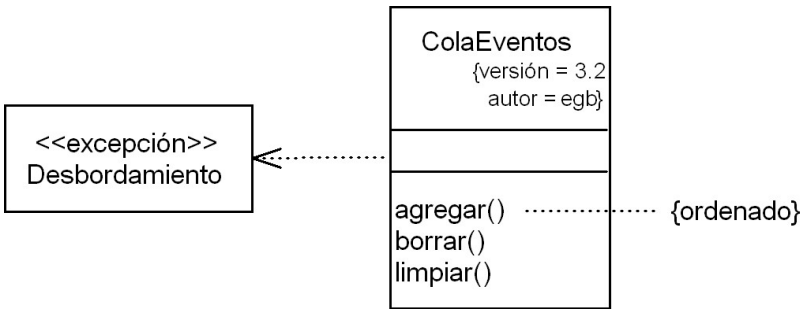


Figura 3.62: Mecanismos de Extensibilidad

Un *valor etiquetado* extiende las propiedades de un bloque de construcción UML, permitiendo crear nueva información en la especificación de ese elemento. Por ejemplo, si se está trabajando en un producto que sufre muchos cambios a través del tiempo, por lo general se quiere mantener la pista de las versiones y autores de ciertas abstracciones críticas. Los conceptos de versión y autor se pueden agregar a cualquier bloque de construcción, como una clase, introduciendo nuevos valores etiquetados a ese bloque. En la Figura 3.62, la clase *ColaEventos* es extendida para mostrar explícitamente su versión y autor.

Una *restricción* extiende la semántica de un bloque de construcción UML, permitiendo el agregado de nuevas reglas o la modificación de las existentes. Por ejemplo,

se podría querer restringir la clase *ColaEventos* de forma tal que todos los agregados se realicen en orden. Como muestra la Figura 3.62, se puede agregar una restricción que marque esto explícitamente para la operación *agregar*.

Colectivamente, estos mecanismos de extensibilidad permiten adaptar el UML a las necesidades de cada proyecto. Estos mecanismos también permiten la adaptación del UML a nuevas tecnologías de software, como la aparición de lenguajes de programación más poderosos. Se pueden agregar nuevos bloques de construcción, modificar la especificación de los existentes, e incluso cambiar sus semánticas. Naturalmente, es importante que esto se haga de forma controlada para que no se pierda el propósito original del UML: la comunicación de información.

### 3.3.3. Arquitectura

Visualizar, especificar, construir, y documentar un sistema de software demanda que el sistema sea visto desde varias perspectivas diferentes. Los distintos interesados observan el proyecto desde diferentes puntos de vista y en diferentes momentos dentro de la vida del proyecto. La arquitectura de un sistema es quizás el artefacto más importante que se puede usar para manejar estos diferentes puntos de vista y así controlar el desarrollo iterativo e incremental de un sistema a lo largo de su ciclo de vida.

La arquitectura es el conjunto de decisiones significativas acerca de

- La organización de un sistema de software.
- La selección de los elementos estructurales y las interfaces que componen el sistema.
- Los comportamientos, especificados en las colaboraciones entre esos elementos.
- La composición de esos elementos estructurales y de comportamiento dentro de subsistemas cada vez mayores.
- El estilo arquitectónico que guía esta organización: los elementos estáticos y dinámicos y sus interfaces, sus colaboraciones, y su composición.

La arquitectura del software no se ve afectada solamente por la estructura y comportamiento, sino también por el uso, funcionalidad, desempeño, elasticidad, reuso, comprensibilidad, restricciones económicas y tecnológicas, y asuntos estéticos.

Como muestra la Figura 3.63, la arquitectura de un sistema de software se puede describir por medio de cinco vistas entrelazadas. Cada vista es una proyección de la organización y estructura de un sistema, enfocada en un aspecto particular de ese sistema.

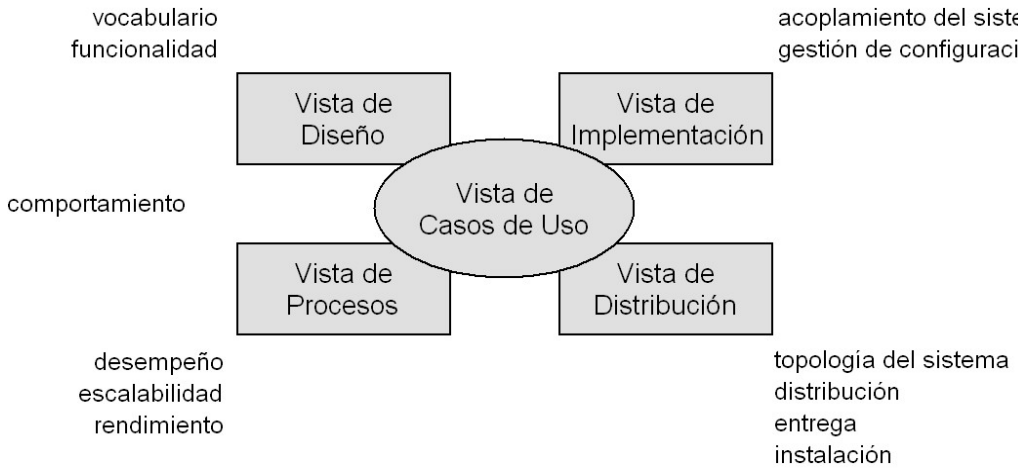


Figura 3.63: Modelado de la Arquitectura de un Sistema

La *vista de casos de uso* de un sistema comprende los casos de uso que describen el comportamiento del sistema como lo ven los usuarios finales, analistas y encargados de las pruebas. Esta vista no especifica realmente la organización de un sistema de software, sino que existe para especificar las fuerzas que dan forma a la arquitectura del sistema. Con el UML, los aspectos estáticos de esta vista son capturados en los diagramas de casos de uso; los aspectos dinámicos son capturados en los diagramas de interacción, de estados, y de actividad.

La *vista de diseño* de un sistema comprende las clases, interfaces, y colaboraciones que forman el vocabulario del problema y de su solución. Esta vista soporta principalmente los requerimientos funcionales del sistema, es decir los servicios que el sistema debería proveer a sus usuarios finales. UML captura los aspectos estáticos de esta vista a través de los diagramas de clases y de objetos, y los dinámicos, a través de los diagramas de interacción, de estados, y de actividad.

La *vista de procesos* de un sistema abarca los hilos y procesos que forman los mecanismos de concurrencia y sincronización del sistema. Esta visión trata el desempeño, escalabilidad, y rendimiento del sistema. UML captura los aspectos estáticos y dinámicos de esta vista con los mismos diagramas que para la vista de diseño, pero

con el foco sobre las clases activas que representan estos hilos y procesos.

La *vista de implementación* de un sistema comprende los componentes y archivos usados para ensamblar y liberar el sistema físico. Esta vista trata principalmente la gestión de configuración de las versiones del sistema, compuesto por componentes y archivos de algún modo independientes que pueden ser ensamblados de diferentes formas para producir un sistema en funcionamiento. Con el UML, los aspectos estáticos de esta vista son capturados por los diagramas de componentes; los aspectos dinámicos son capturados por los diagramas de interacción de estados y de actividad.

La *vista de distribución* de un sistema se compone de los nodos que forman la topología de hardware del sistema sobre la cual se ejecuta el mismo. Esta vista muestra principalmente la distribución, entrega, e instalación de las partes que forman el sistema físico. UML captura los aspectos estáticos de esta vista con los diagramas de distribución, y los aspectos dinámicos con los diagramas de interacción, de estados, y de actividad.

### 3.3.4. Objeciones y Problemas del UML

El Lenguaje Unificado de Modelado se ha convertido en un estándar para la especificación, verificación, visualización y documentación de software. Utilizando las reglas proporcionadas por el estándar, los ingenieros software pueden crear modelos concretos y sin ambigüedades. Los creadores de UML han definido el estándar utilizando la terminología UML: usan elementos de modelado para definir el estándar. Esta forma de representar a UML es conocida como metamodelo de UML.

El metamodelo de UML sirve para que los ingenieros de software puedan verificar la corrección de sus modelos. Debería asumirse, por tanto, que el metamodelo de UML está él mismo libre de errores. Sin embargo, un estudio más detenido del metamodelo de UML ha mostrado que esto no es así. Tras cotejar el metamodelo con todas las restricciones y reglas de correcta formación definidas por el estándar, los autores de [F$^+$03] encontraron 450 errores y los clasificaron en tres grupos diferentes:

1. Elementos no accesibles: representa el problema más importante encontrado en el metamodelo, e implica algunos malentendidos acerca del método.

2. Nombres vacíos: algunas reglas del estándar establecen que dos elementos no pueden tener el mismo nombre. No obstante, el estándar no aclara si pueden

existir dos elementos diferentes sin nombre, que podría ser considerado como el mismo nombre (vacío).

3. Miscelánea: este último grupo de problemas trata de nombres duplicados y asociaciones derivadas.

Este artículo analiza y explica la razón de los errores y presenta algunas sugerencias para corregir lo que los autores piensan que son algunas deficiencias en el actual estándar de UML.

La naturaleza informal de UML, que redunda en un mayor espectro de dominios de aplicación, constituye uno de los principales puntos fuertes de UML, pero también una de sus debilidades más importantes, ya que obliga a los desarrolladores a adoptar interpretaciones particulares de los diagramas para proporcionar aplicaciones como la generación de código, simulación o verificación de los modelos, impidiendo la interoperabilidad entre los mismos [SPHP].

La disparidad de ámbitos de aplicación que se pretende para UML supone una dificultad añadida a la hora de establecer una semántica formal y unificada. Las diversas propuestas para paliar este problema se basan en el concepto de *perfil UML*, que no es más que un mecanismo de extensión estándar que trata de adaptar UML a las necesidades de cada dominio añadiendo una semántica concreta.

Otra limitación importante de UML se refiere a la especificación del comportamiento, definido principalmente mediante *acciones*. Por el momento, la única forma de describir una acción en UML es mediante texto plano sin semántica. Esto da al usuario libertad para describir cualquier cosa de cualquier modo, pero conlleva inconvenientes relacionados con la generación automática de código, la verificación y la simulación. Como solución particular las herramientas interpretan el contenido del texto que describe la acción y emplean a tal efecto lenguajes de programación ya existentes como C/C++, lo que supone tomar decisiones de implementación en las primeras fases del desarrollo.

Además de estas deficiencias de UML y sus problemas asociados, existen otras limitaciones más cercanas al dominio de los sistemas de tiempo real, principalmente relacionadas con:

- Comunicaciones: es posible especificar qué señales recibirá una clase pero no los caminos para la comunicación, o qué ocurre si dos clases pueden tratar la

misma señal.

- Temporización: no existe semántica asociada al concepto de tiempo y no está claro cómo debe ser utilizado.

Otro problema de UML es que no se presta con facilidad al diseño de sistemas distribuidos. En tales sistemas cobran importancia factores como transmisión, serialización, persistencia, etc., y UML no cuenta con maneras de describir tales factores. No se puede, por ejemplo, usar UML para señalar que un objeto es persistente o remoto, o que existe en un servidor que corre continuamente y que es compartido entre varias instancias de ejecución del sistema analizado.

En 1997 Bertrand Meyer escribió un artículo de tono gracioso [Mey97] (en el que un supuesto estudiante escribe a su profesor) llamado "*UML: The positive spin*" en el que desnuda las principales falencias de UML. Más allá de la ironía, los puntos que destaca en contra de UML son:

- Es por demás complejo.
- No es orientado a objetos.
- Carece totalmente de una semántica formal.
- Dificulta extremadamente la reversibilidad.

## 3.4. Métricas Orientadas a Objetos

Las medidas y las métricas son componentes clave de cualquier disciplina de la ingeniería; la ingeniería de software no es una excepción. Lamentablemente, la utilización de métricas para sistemas orientados a objetos ha progresado con mucha más lentitud que la utilización de los demás métodos orientados a objetos. Sin embargo, a medida que los sistemas OO van siendo más comunes, resulta esencial que los ingenieros del software dispongan de mecanismos cuantitativos para estimar la calidad de los diseños y la efectividad de los programas OO.

### 3.4.1. Objetivo de las Métricas Orientadas a Objetos

Los objetivos principales de las métricas orientadas a objetos son los mismos que los existentes para las métricas derivadas para el software convencional. Entre otras

cosas, las métricas permiten [BeAC94]:

- La evaluación de mejoras, suministrando la solución más abarcativa al problema de evaluar los beneficios de la migración o transición tecnológica.
- La reducción de la impredecibilidad asociada con los esfuerzos de desarrollo de software.
- Un control de calidad efectivo.
- Una identificación temprana de problemas ocultos.
- La cuantificación de la extensión y beneficios del reuso.
- El refinamiento de los modelos de estimación para la predicción de atributos externos del proceso de software, como tiempo de desarrollo, esfuerzo de mantenimiento, y tasas de fallos.

Aunque las razones para efectuar mediciones en el desarrollo de sistemas OO parecen ser similares, las métricas de software actuales (Líneas de Código, las de Halstead [Hal77] y McCabe [McC76], etc.) están dirigidas a los lenguajes procedurales tradicionales. El paradigma OO incluye nuevos conceptos y abstracciones, como clases, métodos, mensajes, herencia, polimorfismo, sobrecarga, y encapsulamiento, que no eran abordados en las métricas anteriores. Por lo tanto existe una necesidad creciente de métricas adaptadas al paradigma OO que ayuden a manejar y fomentar la calidad en el desarrollo de software.

Cada uno de los objetivos enumerados es importante en sí, pero para el ingeniero de software, la calidad del producto debe ser lo primordial. ¿Cómo se puede medir la calidad de un sistema OO? ¿Qué características del modelo de diseño se pueden estimar para determinar si el sistema será o no fácil de implementar, se podrá probar, será fácil de modificar, y lo que es más importante, resultará admisible para los usuarios finales? Estas cuestiones se tratan en esta sección.

### 3.4.2. Características distintivas

Las medidas para cualquier producto de la ingeniería están gobernadas por las características únicas de ese producto. El software orientado a objetos es fundamentalmente distinto del software que se desarrolla utilizando métodos convencionales.

Por esta razón, las métricas técnicas para sistemas OO deben ajustarse a las características que distinguen el software OO del software convencional.

Berard [Ber95] define cinco características que dan lugar a unas métricas especializadas: localización, encapsulamiento, ocultamiento de información, herencia y técnicas de abstracción de objetos. Los ejemplos mencionados para cada categoría se verán en detalle en las secciones siguientes.

### Localización

La localización es una característica del software que indica la forma en que se concentra la información dentro de un programa. En el contexto OO, la información se concentra mediante el encapsulamiento tanto de datos como de procesos dentro de los límites de una clase u objeto.

Dado que las clases constituyen la unidad básica de los sistemas OO, la localización está basada en los objetos. Por tanto, las métricas deberían de ser aplicables a la clase (objeto) como si se tratara de una unidad completa. Además, la relación entre operaciones (funciones) y clases no es necesariamente uno–a–uno. Por tanto, las métricas que reflejan la forma en que colaboran las clases deben de ser capaces de adaptarse a las relaciones uno–a–muchos y muchos–a–uno. Ejemplos de estas métricas son Respuesta Para una Clase y Métodos Ponderados por Clase (ver la Sección 3.4.4).

### Encapsulamiento

Para los sistemas OO, el encapsulamiento abarca las responsabilidades de una clase, incluyendo sus atributos y operaciones. También abarca los estados de la clase, según se definen mediante valores específicos de atributos.

El encapsulamiento influye en las métricas modificando el objetivo de la medida, que pasa de ser un único módulo a ser un paquete de datos (atributos) y de módulos de procesamiento (operaciones). Además, el encapsulamiento impulsa a la medida hasta un nivel de abstracción más elevado. Algunos ejemplos de métricas de encapsulamiento son la Carencia de Cohesión en los Métodos, Porcentaje Público y Protegido, y Acceso Público a Datos Miembros (ver la Sección 3.4.6). Compárese este nivel de abstracción con las métricas convencionales, que se centran en un recuento de condiciones booleanas (complejidad ciclomática) o de líneas de código.

### Ocultamiento de Información

El ocultamiento de información suprime (u oculta) los detalles operativos de un componente de un programa. Tan sólo se proporciona la información necesaria para acceder a ese componente a aquellos otros componentes que deseen acceder a él.

Un sistema OO bien diseñado debería de impulsar al ocultamiento de información. Por tanto, aquellas métricas que proporcionen una indicación del grado en que se ha logrado el ocultamiento proporcionarán una indicación de la calidad del diseño OO. Ejemplos: Proporción de Métodos Ocultos y la Proporción de Atributos Ocultos (ver la Sección 3.4.4).

### Herencia

La herencia es un mecanismo que hace posible que las responsabilidades de un objeto se propaguen a otros objetos. La herencia se produce a lo largo de todos los niveles de la jerarquía de clases. En general, el software convencional no admite esta característica.

Dado que la herencia es una característica fundamental de muchos sistemas OO, hay muchas métricas OO que se centran en ella. Entre los ejemplos se cuentan la Proporción de Métodos Heredados, la Proporción de Atributos Heredados, y la Profundidad del Árbol de Herencia (Sección 3.4.4).

### Abstracción

La abstracción es un mecanismo que permite al diseñador centrarse en los detalles esenciales de algún componente de un programa (tanto si es un dato como si es un proceso) sin preocuparse por los detalles de nivel inferior. La abstracción es un concepto relativo. A medida que se asciende a niveles más elevados de abstracción, se ignoran más y más detalles.

Dado que una clase es una abstracción que se puede visualizar con muchos niveles distintos de detalle, y de muchas maneras diferentes (por ejemplo, como una lista de operaciones, como una sucesión de estados, como una serie de colaboraciones), las métricas OO representan las abstracciones en términos de medidas de una clase (por ejemplo, número de instancias por clase por aplicación, número de clases parametrizadas por aplicación, y razón de clases parametrizadas a clases no parametrizadas).

### 3.4.3. Métricas para el Modelo de Diseño OO

Gran parte del diseño orientado a objetos es subjetivo. Un diseñador experimentado "sabe" cómo puede caracterizar un sistema OO para que implemente de forma efectiva los requisitos del cliente. Pero a medida que los modelos de diseño OO van creciendo de tamaño y complejidad, puede resultar beneficiosa una visión más objetiva de las características del diseño, tanto para el diseñador experimentado como para el menos experimentado.

Una visión objetiva del diseño debería de tener un componente cuantitativo, y esto nos lleva a las métricas OO. En realidad, las métricas para sistemas OO se pueden aplicar no sólo al modelo de diseño, sino también al modelo de análisis. En las secciones siguientes se exploran métricas que proporcionan una información de calidad en el nivel de clase OO y en el nivel de operaciones, así como otras métricas que son aplicables para la gestión y comprobación de proyectos.

### 3.4.4. Métricas Orientadas a Clases

La clase es la unidad fundamental de todo sistema OO. Por consiguiente, las medidas y métricas para una clase individual, la jerarquía de clases, y las colaboraciones de clases resultarán sumamente valiosas para un ingeniero de software que tenga que estimar la calidad de un diseño. Todas estas características se pueden utilizar como base para una medida.

#### El Conjunto de Métricas CK

Uno de los conjuntos de métricas de software OO a los que se hace más ampliamente referencia es el propuesto por Chidamber y Kemerer [CK94]. Los autores han propuesto unas métricas de diseño basadas en clases (a las cuales suele aludirse con el nombre de conjunto de métricas CK) para sistemas OO.

***Métodos ponderados por clase (MPC)***

Dada una clase $C$ con los métodos $M_1, M_2, \ldots, M_n$ y $c_1, c_2, \ldots, c_n$ la complejidad de los métodos, MPC se define como la sumatoria de las complejidades de cada método de una clase. Si todos los métodos son considerados de igual complejidad, entonces

$c_i = 1$ y $MPC = n$ (número de métodos).

$$MPC = \sum_{i=1}^{n} c_i$$

Describe la complejidad algorítmica de una clase en términos de las complejidades de todos sus métodos. Está ligada a la calidad de la definición de complejidad de un método ($c_i$). Si esa definición de complejidad se hace en base a una métrica específica (por ejemplo, la Complejidad Ciclomática [McC76]), entonces debería normalizarse para que la complejidad nominal para un método tome valor igual a 1. Algunos autores simplifican esto haciendo $c_i = 1$ para cada método, convirtiéndose así en un simple contador del número de métodos dentro de una clase. En este caso habría que considerarla como una medida del tamaño de una clase y no de complejidad, ya que una clase puede tener pocos métodos pero muy complejos y otra clase puede tener muchos métodos pero muy simples. Puede servir como un indicador de que una clase determinada necesita una descomposición adicional en varias clases.

En la Figura 3.64, si se considera que la complejidad de todos los métodos es igual a 1, se puede decir que para la clase *Figura*, MPC = 3, y en el caso de la clase *Círculo*, MPC = 1.

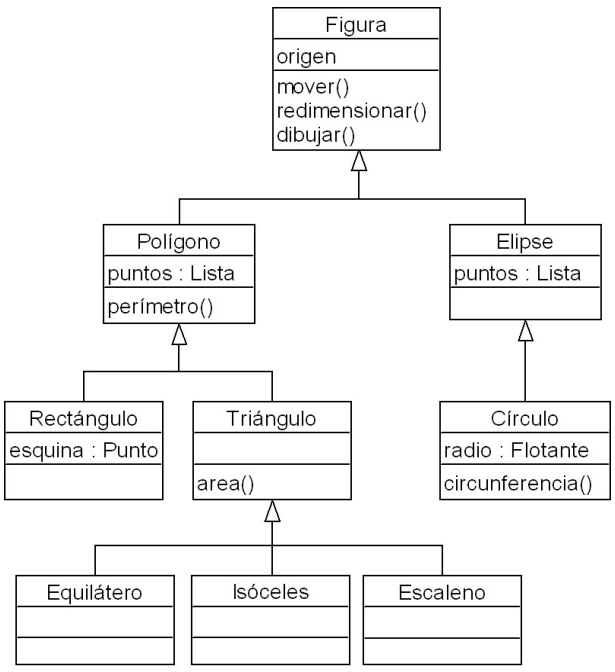


Figura 3.64: Una jerarquía de clases

El número de métodos y su complejidad es un indicador razonable de la cantidad de esfuerzo necesaria para implementar y comprobar una clase. Además, cuanto mayor sea el número de métodos, más complejo será el árbol de herencia (las subclases heredan todos los método de sus predecesoras). Finalmente, a medida que el número de métodos crece para una clase dada, es más probable que se vuelva cada vez más específica de la aplicación, limitando por tanto su potencial de reutilización. Por todas estas razones, MPC debería mantener un valor tan bajo como sea razonable.

Aún cuando podría parecer sencillo desarrollar un contador del número de métodos de una clase, el problema es en realidad más complejo de lo que parece. Para contar métodos, es preciso responder a una pregunta fundamental: ¿Un método pertenece solamente a la clase que lo define, o pertenece también a todas aquellas clases que lo heredan de forma directa o indirecta? Las implicancias de esta cuestión para las métricas pueden ser significativas.

Una crítica que se hace de esta métrica es que en general, cuando una clase aumenta el número y complejidad de sus métodos, su complejidad final tiene un componente sinérgico importante, por lo que reducirla a una suma podría ser una simplificación excesiva. También se argumenta que no hay ni una definición ni una escala de complejidad universalmente aceptadas.

***Profundidad del árbol de herencia (PAH)***

Esta métrica se define como "la longitud desde el último nodo hasta la raíz del árbol" [CK94]. Se trata de la cuenta directa de los niveles en la jerarquía de herencia. En el nivel cero de la jerarquía de encuentra la clase raíz. En la Figura 3.64 el valor de PAH para la jerarquía de clases mostrada es 3.

Chidamber y Kemerer proponen PAH como medida de la complejidad de una clase, la complejidad del diseño y la reutilización potencial, ya que cuanto más profunda se encuentra una clase en la jerarquía, mayor es la probabilidad de heredar más métodos. Es una medida de cuántos ancestros pueden afectar a esta clase.

A medida que PAH crece, es más probable que las clases de niveles inferiores hereden muchos métodos. Esto da lugar a posibles dificultades cuando se intenta predecir el comportamiento de una clase. Una jerarquía de clases profunda (con un valor grande de PAH) lleva también a una mayor complejidad de diseño. Por el lado positivo, los valores grandes de PAH implican que se pueden reutilizar muchos métodos.

Lorenz y Kidd [LK94] sugieren un umbral de 6 niveles como indicador de un abuso

en la herencia tanto en Smalltalk como en C++. En lenguajes como Java o Smalltalk, las clases siempre heredan de la clase Object, lo que añade uno a PAH.

***Número de descendientes directos (NDD)***

Las subclases que son inmediatamente subordinadas a una clase de la jerarquía de clases se denominan sus *descendientes directos*. En la Figura 3.64 la clase *Triángulo* tiene tres descendientes directos (las subclases *Equilátero*, *Isóceles* y *Escaleno*), mientras que la clase *Elipse* tiene sólo uno (la subclase *Círculo*).

A medida que crece el número de descendientes, se incrementa la reutilización, pero también es cierto que a medida que crece NDD la abstracción representada por la clase predecesora puede verse diluida. A medida que NDD va creciendo, la cantidad de pruebas crecerá también.

***Acoplamiento entre clases objeto (ACO)***

ACO es el número de clases a las cuales una clase está ligada, sin tener con ella relaciones de herencia. Hay dependencia entre dos clases cuando una de ellas usa métodos o variables de la otra clase. No se hace diferencia entre invocar una operación de otra clase o cambiar un atributo.

Los autores sugieren que sea un indicador del esfuerzo necesario para el mantenimiento y las pruebas. Cuanto más independiente es un objeto, más fácil es reutilizarlo en otra aplicación. Al reducir el acoplamiento se reduce la complejidad, se mejora la modularidad y se promueve el encapsulamiento. Una medida del acoplamiento es útil para determinar la complejidad de las pruebas necesarias de distintas partes de un diseño. Cuanto mayor sea el acoplamiento entre objetos más rigurosas han de ser las pruebas.

Siguiendo con el ejemplo de la Figura 3.64, todas las clases tienen un valor de ACO = 0, ya que los métodos heredados no se cuentan. En la Figura 3.65, por el contrario, el objeto instancia de la clase *Extracción* tiene un grado de acoplamiento de 3, porque está relacionado con otros 3 objetos de clases diferentes, con las que no tiene relaciones de herencia.

***Respuesta para una clase (RPC)***

El *conjunto de respuesta* de una clase es "un conjunto de métodos que pueden ser ejecutados potencialmente en respuesta a un mensaje recibido por un objeto de esa clase" [CK94]. RPC se define como el número de métodos existentes en el conjunto

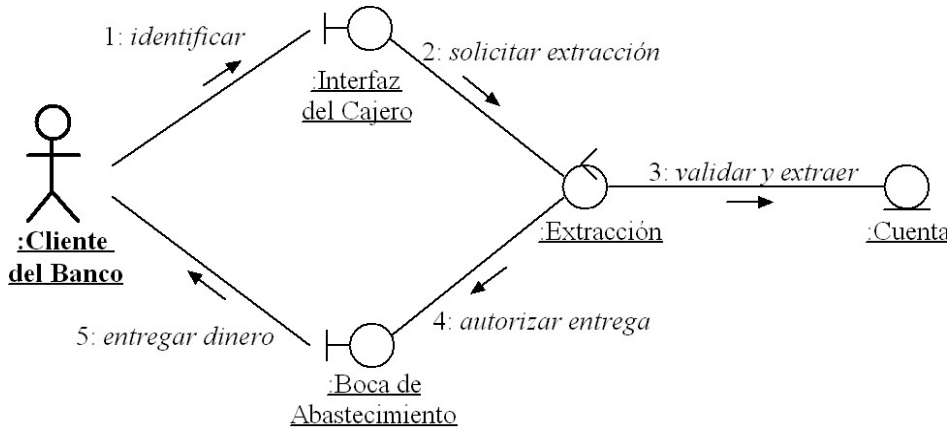


Figura 3.65: Un ejemplo de acoplamiento

de respuesta, es decir, el número de métodos locales a una clase más el número de métodos llamados por los métodos locales.

Para los autores, RPC es una medida de la complejidad de una clase a través del número de métodos y de su comunicación con otras, pues incluye los métodos llamados desde fuera de la clase. Cuanto mayor es RPC, más complejidad tiene el sistema, ya que es posible invocar más métodos como respuesta a un mensaje, exigiendo mayor nivel de comprensión, lo que implica mayor tiempo y esfuerzo de prueba y depuración. Harrison y sus colegas [HCN00] han señalado que la definición de esta métrica es ambigua y fuerza al usuario a interpretarla.

Volviendo al ejemplo de la Figura 3.64, el valor de RPC para la clase *Polígono* surge de sumar la cantidad de clases que pueden invocar al método *perímetro()* (que son 5) más la cantidad de clases que pueden invocar al método *área()* perteneciente a la clase *Triángulo* (que son 3). En conclusión, para la clase *Polígono*, RPC = 8.

***Carencia de cohesión en los métodos (CCM)***

Todo método situado dentro de una clase accede a uno o más atributos (llamados también *variables de instancia*). CCM se calcula como la diferencia entre los pares de métodos que no acceden a atributos comunes y los pares de métodos que sí los tienen. Si esa diferencia es negativa, entonces el valor de CCM = 0.

Por ejemplo, si una clase $C$ tiene tres métodos $M_1$, $M_2$ y $M_3$, y cada uno de ellos accede a los conjuntos de variables de estado $I_1 = \{a, b, c, d, e\}$, $I_2 = \{a, b, e\}$ e $I_3 = \{x, y, z\}$, respectivamente, entonces $I_1 \cap I_2$ es un conjunto no vacío, pero

$I_1 \cap I_3$ y $I_2 \cap I_3$ son conjuntos vacíos. Como se dijo antes, CCM es igual al número de intersecciones vacías menos el número de intersecciones no vacías, que en este caso es igual a uno.

CCM indica la calidad de la abstracción hecha en la clase. Usa el concepto de grado de similitud de métodos. Si no hay atributos comunes, el grado de similitud es cero. Una baja cohesión incrementa la complejidad y por tanto la facilidad de cometer errores durante el proceso de desarrollo. Estas clases podrían probablemente ser divididas en dos o más subclases aumentando la cohesión de las clases resultantes.

Es deseable una alta cohesión en los métodos dentro de una clase, ya que ésta no se puede dividir fomentando el encapsulamiento. [HS96] destaca dos problemas con esta métrica:

- No se dan guías para la interpretación de esta métrica.
- Dos clases pueden tener el mismo valor para CCM, mientras una tiene más variables comunes que la otra. Por ejemplo, la Figura 3.66 muestra dos clases diferentes, donde las distintas $M_i$ son los métodos de cada clase y las $A_i$ son las variables de instancia a las que puede acceder cada método. En ambos casos el cálculo de CCM da como resultado un valor de 8, aunque intuitivamente la clase de la derecha es mucho más cohesiva.

### Métricas Propuestas por Lorenz y Kidd

Lorenz y Kidd [LK94] dividen las métricas basadas en clases en cuatro amplias categorías: tamaño, herencia, valores internos y valores externos. Las métricas orientadas a tamaños para una clase OO se centran en recuentos de atributos y de operaciones para una clase individual, y promedian los valores para el sistema OO en su totalidad. Las métricas basadas en herencia se centran en la forma en que se reutilizan las operaciones a lo largo y ancho de la jerarquía de clases. Las métricas para valores internos de clase examinan la cohesión y asuntos relacionados con el código, y las métricas orientadas a valores externos examinan el acoplamiento y la reutilización. Algunas de las métricas propuestas por Lorenz y Kidd son:

#### *Tamaño de clase (TC)*

El tamaño general de una clase se puede determinar empleando las medidas siguientes:

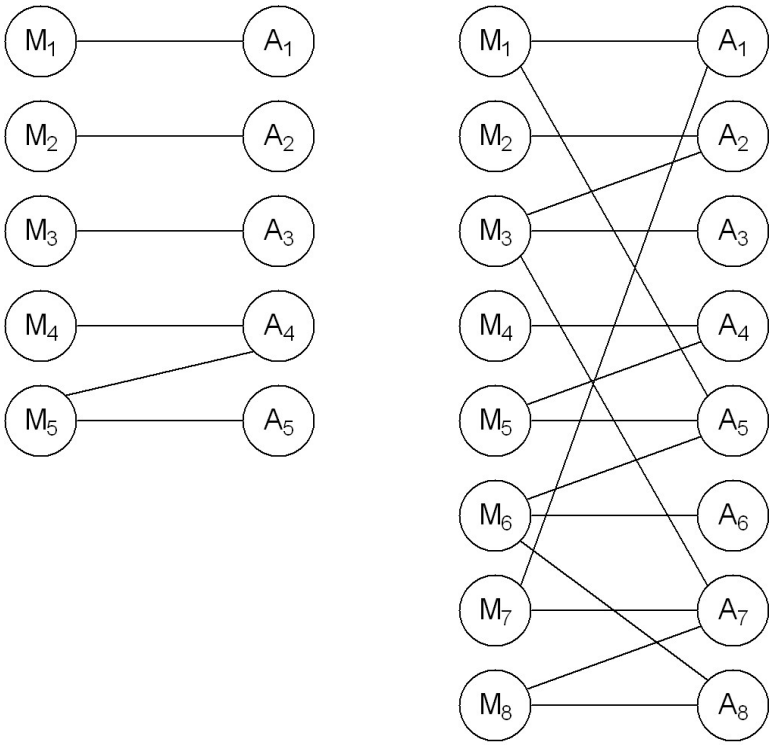


Figura 3.66: Dos clases con CCM = 8

- El número total de operaciones (tanto heredadas como privadas de la instancia) que están encapsuladas dentro de la clase.
- El número de atributos (tanto heredados como privados de la instancia) que están encapsulados dentro de la clase.

Unos valores grandes de TC indican que una clase puede tener demasiada responsabilidad, lo cual reducirá la reusabilidad de la clase y complicará la implementación y la comprobación.

También se pueden calcular los promedios de número de atributos y de operaciones de la clase. Cuanto menor sea el valor medio para el tamaño, más probable es que las clases existentes dentro del sistema se puedan reutilizar ampliamente.

***Número de operaciones invalidadas por una subclase (NOI)***

Existen casos en que una subclase sustituye una operación heredada de su superclase por una versión especializada para su propio uso, y a esto se le denomina *invalidación*. Los grandes valores de NOI suelen indicar un problema de diseño, dado que una subclase debería de ser una especialización de su superclase, y debería

limitarse a extender los servicios (operaciones) de las superclases. Si NOI es elevado, entonces el diseñador ha violado la abstracción implicada por la superclase. Esto da lugar a un jerarquía de clases débil, y a un software OO que puede resultar difícil de comprobar y modificar.

***Número de operaciones añadidas por una subclase (NOA)***

Las subclases se especializan mediante la adición de operaciones y atributos privados. A medida que crece el valor de NOA, la subclase se va alejando de la abstracción implicada por la superclase. En general, a medida que crece la profundidad de la jerarquía de clases, el valor de NOA en los niveles inferiores de la jerarquía debería disminuir.

***Índice de especialización (IE)***

El índice de especialización proporciona una indicación aproximada del grado en que las subclases redefinen el comportamiento de sus superclases. La especialización se puede alcanzar añadiendo o borrando operaciones, o bien por invalidación.

$$IE = \frac{NOI \times nivel}{M_{total}}$$

en donde *nivel* es el nivel de la jerarquía de clases en que reside la clase, y $M_{total}$ es el número total de métodos para la clase.

Esta fórmula pondera más las redefiniciones que ocurren en niveles más profundos del árbol de herencia, ya que, cuanto más especializada es una clase, menos probabilidad existe de que su comportamiento sea reemplazado. Cuando se utilizan *frameworks* (clases de bibliotecas especializadas), algunos métodos deben ser redefinidos: estos métodos no se deben tener en cuenta al calcular esta métrica [E$^+$01].

IE se propone como medida de la calidad en la herencia. IE puede indicar cuando hay demasiados métodos redefinidos, de tal forma que las abstracciones pueden no ser apropiadas y sea necesario reemplazar su comportamiento. Generalmente, una subclase debería extender el comportamiento de la superclase con nuevos métodos más que reemplazar o borrar comportamiento a través de redefiniciones. Lorenz y Kidd sugieren un valor del 15 % para ayudar a identificar superclases que no tienen mucho en común con sus subclases. Cuanto más profundizamos en la jerarquía, más especializada ha de ser la subclase.

### Métricas propuestas por Abreu y Melo

El conjunto de métricas MOOD (Metrics for Object Oriented Design) definido por [BeAM96] opera a nivel de sistema. Se refiere a mecanismos estructurales básicos en el paradigma de la orientación a objetos como encapsulación (MHF y AHF) y herencia (MIF y AIF). En general, las métricas a nivel de sistema pueden derivarse de otras métricas usando métodos estadísticos como la media, etc. Éstas son utilizadas para identificar características del sistema. Este conjunto de métricas es explicado a continuación.

#### *Proporción de Métodos Ocultos (MHF)*

Es el cociente entre los métodos definidos como protegidos o privados y el número total de métodos. Para el cálculo de esta métrica, los métodos heredados no se consideran. MHF se propone como una medida de encapsulación, cantidad relativa de información oculta. Los autores han demostrado empíricamente que cuando se incrementa MHF, la densidad de defectos y el esfuerzo necesario para corregirlos debería disminuir.

#### *Proporción de Atributos Ocultos (AHF)*

Es el cociente entre los atributos definidos como protegidos o privados y el número total de atributos. AHF se propone como una medida de encapsulación.

Idealmente esta métrica debe de ser siempre 100 %, intentando ocultar todos los atributos. Las pautas de diseño sugireren que no se debe emplear atributos públicos, ya que se considera que violan los principios de encapsulación al exponer la implementación de las clases.

#### *Proporción de Métodos Heredados (MIF)*

Se define como el cociente entre la suma de todos los métodos heredados en todas las clases y el número total de métodos (localmente definidos más los heredados) en todas las clases. Sus autores la proponen como una medida de la herencia y como consecuencia, una medida del nivel de reuso. También se propone como ayuda para evaluar la cantidad de recursos necesarios a la hora de testear. El uso de la herencia se ve como un compromiso entre la reusabilidad que proporciona, y la comprensibilidad y mantenimiento del sistema.

#### *Proporción de Atributos Heredados (AIF)*

Se define como el cociente entre el número de atributos heredados y el número total de atributos. Al igual que MIF, AIF se considera un medio para expresar el nivel de reusabilidad en un sistema, aunque demasiado reuso de código a través de herencia hace que el sistema sea más dificil de entender y mantener.

### 3.4.5. Métricas Orientadas a Operaciones

Dado que la clase es la unidad dominante en los sistemas OO, se han propuesto menos métricas para las operaciones de clases. Los métodos tienden a ser pequeños, tanto en términos del número de sentencias como en términos de su complejidad lógica [CS95a], lo cual sugiere que la estructura de conectividad de un sistema puede resultar más importante que el contenido de los módulos individuales.

Sin embargo, existen algunas ideas que pueden llegar a apreciarse examinando las características medias de las operaciones de clases. A continuación se indican tres métricas sencillas propuestas por Lorenz y Kidd [LK94]:

***Tamaño medio de operación ($TO_{avg}$)***

Aún cuando se podrían utilizar las líneas de código como indicador para el tamaño de operación, el *número de mensajes enviados por la operación* proporciona una alternativa para el tamaño de la operación. A medida que crece el número de mensajes enviados por una única operación, es probable que las responsabilidades no hayan sido bien asignadas dentro de la clase.

***Complejidad de operación (CO)***

La complejidad de una operación se puede calcular empleando cualquiera de las métricas de complejidad propuestas para el software convencional [Zus90]. Dado que las operaciones deberían de limitarse a una responsabilidad específica, el diseñador debería de esforzarse por mantener el valor de CO tan bajo como sea posible.

***Número medio de parámetros por operación ($NP_{avg}$)***

Cuanto más grande sea el número de parámetros de la operación, más compleja será la colaboración entre objetos. En general, $NP_{avg}$ debería de mantenerse tan bajo como sea posible.

### 3.4.6. Métricas para Pruebas Orientadas a Objetos

Las métricas de diseño indicadas en las Secciones 3.4.4 y 3.4.5 proporcionan una indicación de la calidad del diseño. También proporcionan una indicación general de la cantidad de esfuerzo de pruebas necesario para aplicarlo en un sistema OO.

Binder [Bin94b] sugiere una amplia gama de métricas de diseño que tienen influencia directa en la "comprobabilidad" de un sistema OO. Las métricas se organizan en categorías que reflejan características de diseño importantes:

#### Encapsulamiento

***Carencia de cohesión en métodos (CCM)***

Cuanto más alto sea el valor de CCM, más estados será preciso probar para asegurar que los métodos no den lugar a efectos colaterales.

***Porcentaje público y protegido (PPP)***

Los atributos públicos se heredan de otras clases, y por tanto son visibles por esas clases. Los atributos protegidos son una especialización y son privados de alguna subclase específica. Esta métrica indica el porcentaje de atributos de clase que son públicos. Unos valores altos de PPP incrementan la probabilidad de efectos colaterales entre clases. Es preciso diseñar comprobaciones que aseguren que se descubran estos efectos colaterales.

***Acceso público a datos miembros (APD)***

Esta métrica indica el número de clases (o métodos) que pueden acceder a los atributos de otra clase, violando así el encapsulamiento. Unos valores altos de APD dan lugar a un potencial de efectos colaterales entre clases. Es preciso diseñar comprobaciones para asegurar que se descubran estos efectos colaterales.

#### Herencia

***Número de clases raíz (NCR)***

Esta métrica es un recuento de las jerarquías de clases distintas que se describen en el modelo de diseño. Es preciso desarrollar conjuntos de pruebas para cada una de las clases raíz, y para la correspondiente jerarquía de clases. A medida que crece NCR crece también el esfuerzo de comprobación.

***Admisión (ADM)***

Cuando se utiliza en el contexto OO, la admisión es una indicación de herencia múltiple. $ADM > 1$ indica que una clase hereda sus atributos y operaciones de más de una clase raíz. Siempre que sea posible, es preciso evitar un valor de $ADM > 1$.

***Número de descendientes (NDD) y profundidad del árbol de herencia (PAH)***

Tal como se describía anteriormente, los métodos de superclase tendrán que ser comprobados de nuevo para cada una de las subclases.

### 3.4.7. Métricas para Proyectos Orientados a Objetos

El trabajo del administrador de un proyecto es planificar, coordinar, seguir y controlar un proyecto de software. ¿Existen entonces métricas OO especializadas que pueda utilizar el administrador del proyecto para disponer de una mejor visión de su progreso? La respuesta es, por supuesto, "sí".

La primera actividad que desarrolla el administrador del proyecto es la planificación, y una de las primeras tareas de la planificación es la estimación. Recuerde el modelo evolutivo de procesos, en donde se vuelve a visitar la planificación tras cada iteración del software. Por tanto, el plan y sus estimaciones de proyecto se visitan después de cada iteración de AOO, DOO, e incluso POO.

Uno de los problemas fundamentales a los que se enfrenta un administrador de proyectos durante la planificación es la estimación del tamaño implementado del software. El tamaño es directamente proporcional al esfuerzo y la duración. Las siguientes métricas OO [LK94] pueden aportar ideas acerca del tamaño del software:

***Número de guiones de escenario (NGE)***

El número de guiones de escenario o de casos prácticos es directamente proporcional al número de clases necesarias para satisfacer los requisitos, al número de estados de cada clase, y al número de métodos, atributos y colaboraciones. NGE es un excelente indicador del tamaño del programa.

***Número de clases clave (NCC)***

Las *clases clave* se centran directamente en el dominio del negocio para el problema en cuestión, y tendrán menor probabilidad de ser implementadas mediante reutiliza-

ción. Por esta razón, unos valores elevados de NCC indican que se encontrará una cantidad notable de trabajo de desarrollo.

***Número de subsistemas (NSUB)***

El número de subsistemas proporciona una idea general de la asignación de recursos, de la planificación (con especial hincapié en el desarrollo en paralelo), y del esfuerzo global de integración.

Las métricas NGE, NCC y NSUB se pueden recoger para otros proyectos anteriores OO, y se pueden relacionar con el esfuerzo invertido en el proyecto como un todo y también con las actividades de proceso individuales (por ejemplo: Análisis, Diseño, Programación y Pruebas Orientadas a Objetos). Estos datos se pueden utilizar también junto con las métricas de diseño descritas anteriormente, con el propósito de calcular "métricas de productividad" tales como el número medio de clases por desarrollador o el promedio de métodos por persona y mes. En su conjunto, estas métricas se pueden utilizar para estimar el esfuerzo, la duración, el personal y otras informaciones acerca del proyecto actual.

# Capítulo 4

# Métodos Formales

## 4.1. Introducción a los Métodos Formales

### 4.1.1. Objetivos de los Métodos Formales

Las computadoras se usan cada vez más en tareas donde una falla puede provocar graves consecuencias, e incluso la pérdida de vidas. La *confiabilidad*, que podríamos describir como que el sistema haga la tarea que se supone que tiene que hacer, es un requerimiento tanto del hardware como del software. Hacer software confiable es especialmente difícil: cualquier programa capaz de hacer algo medianamente interesante es tan complejo que resulta imposible probarlo completamente.

Por otro lado, la creación de software nuevo se lleva a cabo usando un lenguaje de programación elegido, y el leguaje de programación provee medios de expresión altamente organizados y precisamente definidos. Esto constituye una base rigurosa para este último paso en la construcción de software. Sin embargo, la creación de cualquier pieza de software no surge simplemente de la nada. Es imprescindible un razonamiento esencial que preceda al acto de generación del código en sí. En el pasado, la fase del proceso intelectual que precedía la escritura real del código no se apoyaba en formalismos análogos a un lenguaje de programación. A pesar de que esta fase del proceso es más abstracta (es decir, carece de especificidad) y por lo tanto más desafiante mentalmente hablando, se ha tenido que proceder informalmente, sin una guía real para la intuición.

Una técnica importante que apunta al aumento de la confiabilidad del software es el uso de *métodos formales*. La idea básica es que debería ser posible *razonar* sobre

las propiedades del software, o de los sistemas que incluyen software.

Mientras que la intuición nunca pierde su lugar, se pretende que los métodos formales provean los medios para lograr una mayor precisión al idear y documentar esta etapa preliminar del proceso de creación de software. Cuando se hace bien, esto puede beneficiar en todos los aspectos de la creación del software: formulación de requerimientos del usuario, implementación, verificación/prueba, y la creación de documentación.

Los métodos formales se interesan principalmente en la *especificación* del software, y en asuntos directamente relacionados con ella. Es decir, en desarrollar una declaración precisa de *qué* debe hacer el software, y evitar restricciones explícitas (o incluso implícitas) de *cómo* se debe hacer. Se intenta proveer una descripción completamente rigurosa de los *resultados* que debe devolver un elemento de software, mientras se les deja a los programadores toda la flexibilidad para usar su ingenio y obtener esos resultados. Como se pretende que estas descripciones de resultados precedan a la construcción del software y su ejecución por hardware, lograr precisión y rigor requiere el uso de formalismos matemáticos/lógicos. Estas especificaciones independientes de la plataforma sirven como un contrato técnico inicial entre el programador y el cliente, y luego guían la creación, verificación, y documentación del software.

Los métodos formales no son la respuesta completa. Para empezar, las "demostraciones" pueden contener defectos. ¿Cómo se sabe si el modelo del mundo contenido en el software se refleja en el mundo real? Los métodos formales pueden ofrecer confianza de que los requerimientos son correctos, encontrando inconsistencias y omisiones, pero confianza no es certeza. Además, el uso de métodos formales implica interpretar los requerimientos mediante la creación de un modelo de ellos en otro lenguaje. Tal proceso es propenso a errores.

Los métodos formales no reemplazan a las pruebas, y deben aplicarse con la debida atención al control de calidad y a la efectividad del costo. Pero son invalorables para mejorar la confiabilidad.

### 4.1.2. Rol de los Métodos Formales

#### La importancia del software de alta calidad

El desarrollo de software es una actividad vital en la sociedad moderna, y es probable que su importancia aumente en el futuro [Fle06]. El software maneja nuestras cuentas bancarias, paga nuestros salarios, controla los aviones en los que volamos, regula la generación y distribución de energía, controla nuestras comunicaciones, etc.

#### Características del software de alta calidad

El software de alta calidad comparte los siguientes atributos obvios:

- Es intuitivo y fácil de usar – las cosas correctas ocurren "automáticamente".
- Es eficiente – la gente usa computadoras para que las cosas se hagan rápido.
- Sobre todo, es correcto – siempre produce los resultados anunciados y no sufre caídas.

#### La necesidad de precisión en la especificación del software

La noción de que los componentes de software se pueden reusar es una motivación principal de la programación orientada a objetos, y virtualmente se ha convertido en un postulado de la programación. Para reusar un componente de software previamente escrito (o crear uno nuevo), un ingeniero de software debe tener una descripción precisa de su comportamiento. Esta precisión es esencial, ya que incluso una mínima interpretación errónea de la función de un componente que no sea advertida al principio puede causar serios errores, que son difíciles y caros de corregir más tarde en el proceso.

#### El rol de los métodos formales

Se pretende que los métodos formales sistematicen e introduzcan rigor en todas las fases del desarrollo de software. Esto ayuda a evitar el olvido de cuestiones críticas, provee un medio estándar para registrar suposiciones y decisiones varias, y forma una base para la consistencia entre muchas actividades relacionadas. Al brindar mecanismos de descripción precisos y no ambiguos, los métodos formales facilitan la

comprensión requerida para combinar las distintas etapas del desarrollo de software en un esfuerzo exitoso.

El lenguaje de programación usado para el desarrollo del software provee una sintaxis y semántica precisas para la fase de implementación, y esto ha sido así desde que se comenzó a escribir programas. Pero la precisión en todas las demás etapas de desarrollo de software debe derivar de otras fuentes. El término "métodos formales" se aplica a una amplia colección de formalismos y abstracciones ideadas para soportar un nivel comparable de precisión para otras etapas del desarrollo de software. Mientras esto incluye asuntos actualmente bajo desarrollo activo, varias metodologías han alcanzado un nivel de madurez que puede ser de beneficio para los prácticos.

### 4.1.3. Algunos motivos para estudiar métodos formales

La industria de software tiene una larga y bien ganada reputación de no cumplir con lo que promete. En [Gib94] se muestra una buena cantidad de ejemplos, y se observa que "a pesar de 50 años de progreso, la industria de software aún está a años –o quizás a décadas– de la disciplina madura necesaria en una sociedad en la era de la información". Luqi y Goguen [LG97] citan fallas asombrosas en la estimación de costos de desarrollo de software. Goguen [Gog] también llama la atención sobre varias fallas muy notorias: la cancelación de un contrato de IBM por U$D 8.000 millones con la FAA de los EEUU para un nuevo sistema de control aéreo de alcance nacional, la cancelación por parte del Departamento de Defensa de los Estados Unidos de un contrato de U$D 2.000 millones con la IBM para modernizar sus sistemas de información, la falla del software en entregar datos deportivos en tiempo real en las Olimpíadas de 1996, la demora de un año y medio en el sistema automático de manejo de equipaje de United Airlines en el nuevo aeropuerto de Denver, con un costo de U$D 1.1 millones por día, y la lista podría continuar. Peter Neumann [Neu95] revela que tales problemas no son para nada nuevos, aunque parecen estar aumentando. Neumann incluso señala algunas muertes resultantes de sobredosis de radiaciones en un sistema computarizado de radioterapia a mediados de los 80s.

Está claro que no hay precio que pueda asegurar el éxito de los proyectos de software con la tecnología actual. Para proyectos grandes y complejos, un enfoque *ad-hoc* ha resultado inadecuado. La falta de formalidad en lugares clave hace que la ingeniería de software sea demasiado sensible a las debilidades inevitables en las actividades

altamente técnicas y detalladas asociadas con la creación de software. Es esencial contar con apoyo para lograr precisión y control cruzado, y este es precisamente el objetivo de los métodos formales.

### 4.1.4. Introducción a la especificación de programas

Un programa describe un cómputo; su propósito es ser ejecutado en una computadora y llevar a cabo ese cómputo. Los programas se escriben en un lenguaje de programación precisa y formalmente definido, como por ejemplo Java, C, Pascal. La especificación de un programa describe los resultados que se espera que un programa genere; su propósito principal es que se comprenda sin ser ejecutado. Las especificaciones proveen las bases para la metodología de programación.

Una especificación es un contrato técnico entre un programador y su cliente, y se espera que les provea un entendimiento mutuo del programa. Un cliente usa la especificación como guía de uso del programa; un programador la usa como una guía para la construcción del programa. Una especificación completa puede engendrar sub-especificaciones, cada una de las cuales describe un sub-componente del programa. La construcción de estos sub-componentes puede delegarse a otros programadores, de forma que un programador a un nivel también puede ser cliente a otro nivel.

La experiencia muestra que es extremadamente difícil crear software de alta calidad. Una especificación que se comprenda con claridad es un ingrediente vital en el proceso de creación. El lenguaje natural es demasiado vago y ambiguo como para confiar en él para lograr la precisión necesaria. Por supuesto, el programa mismo determina qué resultados se producen, por lo que puede considerarse como la especificación final. Sin embargo, el programa recién está disponible al final de la interacción programador-cliente, y no puede ser un factor para desarrollar un contrato inicial adecuado. Y aún cuando esté completo, el exceso de detalles en los programas dificulta un claro entendimiento y hace que el programa no sirva a los propósitos de una especificación.

El propósito de una especificación requiere que todas las partes tengan plena confianza en las propiedades de los resultados que ésta garantiza. El foco de una especificación debería estar en qué se va a lograr, y no en cómo se va a lograr. Los detalles generados en el programa mismo se deben evitar. Las convenciones precisas y formalmente definidas para escribir especificaciones son una invención mucho más

reciente que los lenguajes de programación. La necesidad de ser completamente precisos sobre los resultados antes de comenzar con el proceso de programación ha llevado a confiar en conceptos basados en las matemáticas (o más precisamente en la lógica).

Como una especificación provee un contrato técnico, es natural basar tanto la construcción como la verificación de un programa en su especificación. Como resultado, los métodos formales incluyen: (1) elementos conceptuales para el desarrollo de especificaciones precisas que puedan servir de guía a la actividad de programación, (2) los medios para utilizar una especificación formal para una verificación rigurosa del programa cuando éste está completo, y (3) la integración de estas ideas en un "sistema de especificación" que puede apoyarse en herramientas computarizadas que asistan a todo el proyecto.

### 4.1.5. Creencias y lineamientos sobre los Métodos Formales

Los trabajos más citados sobre aspectos filosóficos de los métodos formales en la ingeniería de software son [Hal90], [BH95a] y [BH95b]. En los dos primeros se analizan en detalle algunas creencias generalizadas respecto de los métodos formales que resultan no ser completamente ciertas. Como se verá, esas creencias pueden estar relacionadas con ventajas "milagrosas" que en realidad no son tales, con dificultades que en realidad no existen, o con la aplicación y aplicabilidad de los métodos formales. En tanto, en el tercer trabajo mencionado se establecen diez "mandamientos" sobre los métodos formales, o lineamientos a tener en cuenta al momento de aplicar alguno de estos métodos.

Se podría elegir ignorar los métodos formales, si fuesen irrelevantes en la construcción de software de calidad, pero la experiencia demuestra lo contrario. De todos modos, la incertidumbre en el significado y rol de los métodos formales es muy grande. Las discusiones en las referencias citadas intentan identificar y esclarecer algunas observaciones comunes en este sentido.

#### Siete Mitos sobre los Métodos Formales

Los métodos formales son controvertidos. Sus defensores dicen que pueden revolucionar el desarrollo. Sus detractores piensan que son demasiado difíciles. Mientras tanto, para la mayoría de las personas, los métodos formales son tan poco familiares

que es difícil juzgar las afirmaciones opuestas. No hay demasiada evidencia publicada que apoye un lado o el otro, y mucho de lo que se dice sobre los métodos formales se basa en dichos y no en hechos. Así, algunas de las creencias han sido exageradas y han adquirido casi el nivel de mito [Hal90].

El significado del diccionario que parece reflejar mejor la intención de la palabra "mito" en los títulos (y contenido) de los trabajos citados más arriba es "una ficción o media verdad". Presumiblemente, este significado es por lo tanto también precisamente definido como "una falsedad o media mentira". Desde esta perspectiva, los mitos (si están bien categorizados) pueden verse como una lista de aspiraciones y limitaciones de los métodos formales. Los siete mitos sobre los métodos formales que más prevalecen son variantes de los siguientes:

***Mito 1.*** *Los métodos formales pueden garantizar que el software es perfecto.*

El mito más importante es que los métodos formales son de alguna manera todopoderosos, si sólo los mortales pudiesen aplicarlos. Este es un mito perjudicial, porque lleva tanto a expectativas poco realistas como a la idea de que los métodos formales son de alguna manera todo o nada. La realidad es que no se puede dar tal garantía, pero la utilidad de los métodos formales no depende de tal perfección absoluta.

La verdad es que los métodos formales no son infalibles. Debería ser demasiado obvio como para tener que decirlo, pero nada puede lograr la perfección. Lamentablemente, a veces parece que los defensores de los métodos formales dicen que éstos ofrecen una garantía absoluta que no se puede lograr de otra forma. Si se toma esta posición, entonces cualquier problema con el software desarrollado formalmente es una refutación de la utilidad de los métodos formales.

Es importante comprender las limitaciones intrínsecas de los métodos formales. Su falibilidad es la limitación más fundamental, y surge de dos hechos:

- *Algunas cosas nunca se pueden probar.* En software, existen límites en las técnicas de modelado. Primero, los modelos sólo cubren ciertos aspectos del comportamiento de un programa. Segundo, la correspondencia entre la descripción formal y el mundo real es limitada.

  Hay buenos modelos matemáticos para el comportamiento de los programas secuenciales. También existen modelos para comportamiento concurrente, aunque son más difíciles de usar. Algunos dicen que no se pueden modelar formalmente

restricciones de tiempo; esto no es estrictamente correcto, pero es cierto que no se sabe cómo usar esos modelos para desarrollar software que cumpla con las restricciones. Finalmente, aún no es posible modelar propiedades no funcionales, como desempeño, confiabilidad, mantenibilidad, y disponibilidad.

La correspondencia entre los modelos formales de los programas y el comportamiento real de los sistemas está limitado por tres factores: el comportamiento del lenguaje de programación, el sistema operativo, y el hardware subyacente. Para sistemas críticos de seguridad, estas limitaciones son cruciales y no se puede asumir que un programa es correcto sólo porque ha sido probado.

- *Se pueden cometer errores al probar aquellas cosas que sí se pueden probar.* Incluso dentro de los formalismos, se pueden cometer errores al realizar las pruebas, así como se pueden cometer errores al escribir programas. De hecho, las especificaciones formales publicadas contienen errores.

A pesar de estos aparentes problemas, los métodos formales *funcionan*. Hay dos razones: una es que hay algunas formas en que los métodos formales ofrecen garantías cualitativamente diferentes y mejores que cualquier otro método. La otra es que aún cuando los métodos formales no impiden cometer errores, son mucho mejores para dejar en evidencia esos errores.

***Mito 2.*** *Los métodos formales funcionan probando que los programas son correctos.*

En los Estados Unidos, mucho del trabajo en métodos formales se ha concentrado en la verificación de programas. Esto ha hecho que los métodos formales parezcan muy difíciles y poco relevantes en la vida real. Sin embargo, se puede lograr mucho sin ningún tipo de prueba formal.

La realidad es que los métodos formales tienen que ver con las especificaciones. Se usa el término "métodos formales" para referirse al uso de las matemáticas en el desarrollo de software. Las principales actividades que se incluyen son:

- escribir una especificación formal
- probar propiedades sobre la especificación
- construir un programa manipulando matemáticamente la especificación
- verificar un programa mediante argumentos matemáticos

La verificación de programas es sólo un aspecto de los métodos formales. Muchas veces es el más difícil. Para proyectos cuya seguridad no es crítica, la verificación de programas está lejos de ser el aspecto más importante de un desarrollo formal. Como el costo de eliminar errores aumenta dramáticamente a medida que el proyecto progresa, es más importante poner mucha atención en las etapas tempranas.

***Mito 3.*** *Sólo los sistemas altamente críticos se benefician con el uso de los métodos formales.*

Esta creencia se basa en la dificultad percibida de usar métodos formales. La verdad es que los sistemas críticos sí demandan el uso más cuidadoso de los métodos formales, pero en general cualquier sistema se beneficia con el uso de al menos algunas técnicas formales.

Probablemente, las aplicaciones prácticas más grandes de los métodos formales hayan sido en proyectos no críticos. Los métodos formales deberían usarse siempre que el costo de una falla sea alto.

La aplicación de métodos formales puede beneficiar muchas áreas, como la adaptabilidad al propósito, la mantenibilidad, la facilidad de construcción, y una mejor visibilidad. Partiendo de una especificación formal, el proceso de desarrollo puede hacerse muy riguroso si se hace en pequeños pasos, expresando y justificando formalmente cada uno de ellos. También puede hacerse menos riguroso si los pasos son más grandes y sólo se justifican informalmente. Se elige el grado de rigor que mejor se adapte a la aplicación. Si el sistema es crítico, debe desarrollarse en forma completamente formal, por supuesto.

Sin embargo, muchos beneficios de los métodos formales provienen de la etapa de especificación. Así, en un sistema no crítico, incluso si ninguno de los otros pasos es formal, escribir una especificación formal es una gran mejora sobre otros métodos informales.

***Mito 4.*** *Los métodos formales involucran matemáticas complejas.*

Los métodos formales se basan en las matemáticas, y muchas personas creen que esto los hace demasiado difíciles para los ingenieros de software. Este mito se basa a su vez en la visión de que las matemáticas son intrínsecamente difíciles. Pero la realidad es que la matemática para especificación, al menos, es aprendida y usada con facilidad.

Una vez que se reconoce que la práctica de los métodos formales tiene más que ver con escribir especificaciones, las dificultades matemáticas dejan de ser tan importantes. Se pueden desarrollar especificaciones con matemática muy directa, que cualquier ingeniero practicante debería conocer.

La especificación de un problema es más corta y mucho más fácil de entender que su expresión en un lenguaje de programación. La gente les teme a los símbolos nuevos, pero los símbolos matemáticos se introducen para hacer más fácil la matemática, y no para dificultarla. La gente se familiariza rápidamente con los símbolos nuevos.

Esto no significa que escribir especificaciones sea una tarea sencilla. Luego de aprendida la notación, aún quedan dificultades. La principal es hacer las conexiones adecuadas entre el mundo real y el formalismo matemático. Puede ser difícil elegir qué cosas del mundo real se deben modelar - obtener el nivel correcto de abstracción. Algunos programadores ponen demasiado detalle en sus especificaciones y las hacen muy complicadas, o también puede ocurrir lo contrario, que se escriban especificaciones que son demasiado abstractas. Pero estos problemas no son introducidos por la formalidad, sino que son comunes a cualquier tipo de especificación.

***Mito 5.*** *Los métodos formales aumentan el costo de desarrollo.*

Solía decirse que, aunque el uso de métodos formales era muy caro, valía la pena debido al menor costo de mantenimiento del software resultante. Pero es un argumento muy difícil de vender a directores de proyecto bajo presión, cuyo presupuesto es para desarrollo y no para mantenimiento. De hecho, hay alguna evidencia de que el desarrollo puede ser más barato cuando se usa una especificación formal.

Un desarrollo completamente formal, que incluya la comprobación de cada paso de desarrollo, es muy caro, probablemente infactible para cualquier aplicación que no sea crítica. Pero como muchos beneficios provienen sólo de escribir especificaciones formales, es importante saber si esto también es costoso.

La experiencia acumulada en el costo de proyectos que usaron especificaciones formales sugiere que estos son más bajos que los costos de los proyectos que usaron otros métodos. También es importante mencionar que se introducen cambios en el ciclo de vida, es decir que se modifica la forma del proyecto. Se consume más tiempo en la fase de especificación, pero las etapas de implementación, integración y pruebas son más cortas.

***Mito 6.*** *Los métodos formales son inaceptables para los usuarios.*

Una especificación formal está llena de símbolos matemáticos, que la hacen incomprensible para cualquiera que no esté familiarizado con la terminología. Por lo tanto, se supone, una especificación formal es inútil para usuarios no matemáticos.

Sin embargo, la matemática no es la única parte de una especificación formal, sino que soporta muchas otras formas de expresar la especificación, que le dan al cliente un mejor entendimiento en las primeras etapas del proyecto. Lo cierto es que las especificaciones formales ayudan a los usuarios a comprender lo que están obteniendo.

La especificación captura lo que el usuario quiere *antes* de que se construya. Pero para materializar este beneficio, la especificación se debe hacer comprensible para el usuario. Hay tres formas de hacerlo:

- Parafrasear la especificación en lenguaje natural. Esto es esencial. Una especificación matemática debe ir acompañada de una descripción en lenguaje natural que explique qué significa la especificación en términos del mundo real.

- Demostrar las consecuencias de la especificación. Las especificaciones formales son útiles para demostrar mediante razonamiento formal que las especificaciones cumplen con ciertos requerimientos. Se pueden extraer algunas consecuencias de la especificación y presentárselas al cliente.

- Animar la especificación. Esto brinda una capacidad de prototipado inmediata para explorar distintas cuestiones de la especificación.

***Mito 7.*** *Nadie usa métodos formales para proyectos reales grandes.*

Los métodos formales se asocian con frecuencia a departamentos académicos y organizaciones de investigación. Se piensa que sólo esas organizaciones tienen la habilidad necesaria para usarlos, y que los métodos formales sólo son adecuados para las aplicaciones idealizadas que tales grupos llevarían a cabo. Sin embargo, la experiencia está haciendo que este punto de vista se transforme en mito.

La realidad es que los métodos formales se usan diariamente en proyectos industriales. Varias organizaciones usan métodos formales en proyectos de escala industrial. Mucha gente conoce aplicaciones en el área de seguridad, pero el alcance de los métodos formales es mucho más amplio. Procesamiento de transacciones, herramientas

de software, compiladores, control de reactores, hardware, son ejemplos de tipos de proyectos que usan métodos formales.

### Siete Mitos más sobre los Métodos Formales

En esencia, un método formal es una técnica basada en la matemática para describir un sistema. El uso de métodos formales permite especificar, desarrollar, y verificar sistemáticamente un sistema. Sin embargo, las definiciones básicas de métodos formales y términos relacionados son de alguna manera confusas, a pesar de que algunas de sus aplicaciones más publicitadas han hecho posible que una audiencia más amplia los conozca [BH95a].

Incluso términos básicos como "especificación formal" son todavía propensos a confusión. Por ejemplo, las siguientes definiciones alternativas están dadas en un glosario publicado por el IEEE:

1. Una especificación escrita y aprobada de acuerdo con estándares establecidos.

2. Una especificación escrita en notación formal, con frecuencia para su uso en prueba de correctitud.

Aunque la segunda es aceptada en la comunidad de los métodos formales, la primera puede tener más aceptación en los círculos industriales.

Lo que *sí* es claro es que aún poca gente comprende exactamente qué son los métodos formales o cómo se aplican. Muchos no formalistas parecen creer que los métodos formales son meros ejercicios académicos que no tienen relación con problemas del mundo real.

Muchos de los mitos de Hall [Hal90] fueron –y todavía son, en cierta medida– propagados por los medios. Por fortuna, hoy estos mitos son sostenidos más por el público y la comunidad de ciencias de la computación en general que por los desarrolladores de sistemas. Sin embargo, se están propagando nuevos mitos, y de forma más alarmante debido a que están recibiendo una cierta aceptación tácita de la comunidad de desarrollo de sistemas.

***Mito 8.*** *Los métodos formales retrasan el proceso de desarrollo.*

Varios proyectos que utilizaron métodos formales han sobrepasado notoriamente el tiempo planificado. Pero asumir que esto es un problema inherente a los métodos

formales es irracional. Estos proyectos se retrasaron no porque los especialistas en métodos formales carecieran de habilidad, sino debido a su falta de experiencia en determinar cuánto tiempo llevaría el desarrollo.

Se han desarrollado varios modelos que cubren la estimación de costos y tiempo de desarrollo. El más famoso de ellos quizás sea el modelo Cocomo de Boehm [Boe81]. Pero cualquiera de estos modelos debe basarse en información histórica y detalles tales como niveles de experiencia y familiaridad con el problema. Incluso con los métodos de desarrollo tradicionales, esta información no siempre está disponible. Información histórica acerca de proyectos que usaron técnicas formales de desarrollo es incluso más escasa, ya que todavía no se aplicaron métodos formales a un número suficiente de proyectos. Sondeos de desarrollos formales, con especial énfasis en los éxitos, fallas, obstáculos, etc., eventualmente proveerán la información requerida.

Pero a pesar de estas dificultades, varios proyectos con métodos formales han sido muy exitosos, ya que el tiempo de desarrollo fue mucho menor que el estimado, o hubo un significativo ahorro en el costo de desarrollo.

***Mito 9.*** *Los métodos formales carecen de herramientas.*

El soporte de herramientas se ve como una forma de incrementar la productividad y precisión en el desarrollo formal. Muchos proyectos ponen gran énfasis en el soporte de herramientas. Esto no es casual, sino que sigue una tendencia que se espera que resulte en entornos integrados que soporten especificaciones formales.

Varios métodos formales incorporan soporte de herramientas dentro del método mismo. En esta categoría hay lenguajes de especificación con subconjuntos ejecutables, como OBJ [GM96], y métodos formales que incorporan probadores de teoremas como un componente clave, como Larch [GH93] (con el Larch Prover), Nqthm [BM88] (sucesor del probador Boyer-Moore), y más recientemente el Sistema de Verificación de Prototipos PVS [ORS92].

***Mito 10.*** *Los métodos formales reemplazan los métodos de diseño tradicionales de la ingeniería.*

Una de las mayores críticas de los métodos formales es que no son tanto "métodos" como sistemas formales. Aunque proveen soporte para una notación formal (lenguaje de especificación formal), y alguna forma de aparato deductivo (sistema de prueba), no soportan muchos de los aspectos metodológicos de los métodos de desarrollo

estructurado más tradicionales.

En el contexto de una ingeniería, un *método* describe cómo se va a conducir un proceso. En el contexto de la ingeniería de sistemas, un método consiste en un modelo de desarrollo subyacente; un lenguaje o un conjunto de lenguajes; pasos definidos y ordenados; y una guía para aplicar éstos de manera coherente.

Muchos de los métodos llamados formales no abordan todos estos temas. Aunque soportan algunos de los principios de diseño de los métodos más tradicionales –como diseño *top-down* y refinamiento por pasos– ponen poco énfasis en el modelo de desarrollo subyacente y proveen poca guía sobre cómo debería proceder el desarrollo. Los métodos de desarrollo estructurados, que usan un modelo como el espiral de Boehm, por el contrario, en general soportan todas las etapas del ciclo de vida del sistema, desde la captura de requerimientos hasta el mantenimiento post implementación. Por lo general, estos modelos subyacentes reconocen la naturaleza iterativa del desarrollo de sistemas. Sin embargo, muchos métodos de desarrollo formal asumen que a la especificación le sigue el diseño y luego la implementación, en secuencia estricta. Esta esa una visión poco realista del desarrollo –cualquier desarrollador de sistemas complejos debe volver a los requerimientos y a la especificación en etapas muy posteriores del desarrollo.

Aunque Hall refuta el mito de que los métodos formales son inaceptables para los usuarios y requieren una habilidad matemática especial, los métodos de diseño más tradicionales sobresalen en la captura de requerimientos y en la interacción con los usuarios. Ofrecen notaciones entendibles por los no especialistas y sirven de base para un contrato.

Los métodos estructurados tradicionales están severamente limitados debido a que ofrecen pocas formas de razonar sobre la validez de una especificación o si ciertos requerimientos son mutuamente excluyentes. Lo primero sólo se descubre luego de la implementación; lo segundo, durante la implementación. Los métodos formales, por supuesto, dan la posibilidad de razonar sobre los requerimientos, su completitud, y sus interacciones.

De hecho, en lugar de que los métodos formales reemplacen los métodos de diseño tradicionales, una importante área de investigación es la *integración* de los métodos estructurados y formales [Kro93], [Sem92]. Tal integración lleva a un método de desarrollo "verdadero" que soporta completamente el ciclo de vida del software y

permite a los desarrolladores usar técnicas más formales en las fases de especificación y diseño, soportando refinamiento del código ejecutable y propiedades de prueba. El resultado es que se presentan dos vistas del sistema, dejando que los desarrolladores se concentren en los aspectos que les interesan.

Los enfoques para la integración de métodos varían desde la ejecución en paralelo de los métodos estructurado y formal, hasta la especificación formal de transformaciones de notaciones del método estructurado a un lenguaje de especificación formal. Si bien se ha reportado bastante éxito con la primera técnica, el problema surge porque las dos metodologías están siendo abordadas por diferentes personas, y por lo tanto la probabilidad de que se pongan de relieve los beneficios es baja. En muchos casos, los dos equipos de desarrollo no interactúan de manera adecuada. Se han emprendido algunos intentos de integración del segundo tipo, pero si bien éstos pueden tener un gran potencial, aún no se han aplicado a sistemas reales.

***Mito 11.*** *Los métodos formales sólo se aplican al software.*

Los métodos formales se pueden aplicar tanto al desarrollo de software como al diseño de hardware. De hecho, esta es una de las motivaciones del probador de teoremas HOL que se usó para verificar partes del microprocesador Viper. Este es sólo uno de los muchos sistemas de prueba de teoremas que se han aplicado a la verificación de hardware. La prueba de modelos también es importante para probar diseños de hardware.

Otro lenguaje utilizado por ingenieros para diseñar circuitos digitales es el VHDL, definido por el IEEE (*Institute of Electrical and Electronics Engineers*) (ANSI/IEEE 1076-1993). VHDL es el acrónimo que representa la combinación de VHSIC y HDL, donde VHSIC es el acrónimo de *Very High Speed Integrated Circuit* y HDL es a su vez el acrónimo de *Hardware Description Language*.

En [HG92] puede encontrarse un conjunto de artículos escritos por expertos, que cubren en más detalle estos temas.

***Mito 12.*** *Los métodos formales son innecesarios.*

Esto no es cierto. Si bien en algunas ocasiones los métodos formales pueden resultar excesivos, en otras situaciones son muy deseables. De hecho, el uso de métodos formales es recomendable en cualquier sistema donde la correctitud sea una preocupación. Esto se aplica claramente a sistemas críticos de seguridad, pero también a

sistemas en los que se necesita (o se quiere) asegurar que se evitarán las consecuencias catastróficas de una falla.

A veces los métodos formales no son sólo deseables, sino que también son requeridos. Muchos organismos de estándares no sólo usaron lenguajes de especificación formal para hacer sus propios estándares no ambiguos, sino que también mandaron o recomendaron fuertemente el uso de métodos formales en ciertas clases de aplicaciones.

Se crea o no que los métodos formales son necesarios en el desarrollo de sistemas, no se puede negar que de hecho son *requeridos* en ciertas clases de aplicaciones, y probablemente sean requeridos con más frecuencia en el futuro.

***Mito 13.*** *Los métodos formales no son soportados.*

Hace bastante tiempo, el desarrollo formal puede haber sido una actividad o pelea solitaria. Sin embargo hoy en día, el soporte para los métodos formales es incuestionable. El interés de los medios en los métodos formales ha crecido en forma notable, a pesar de que lo hizo desde una pequeña base. Junto con la orientación a objetos, los métodos formales se han transformado rápidamente en palabras comunes en la industria de la computación. Lejos quedaron los días en que investigadores solitarios trabajaban en el desarrollo de notaciones y cálculos adecuados. El desarrollo de los métodos formales más populares le debe mucho a la contribución de muchas personas más allá de los que le dieron origen al método. En muchos casos, los investigadores y prácticos extendieron los lenguajes para soportar sus necesidades particulares, agregando operadores y estructuras de datos útiles y extendiendo los lenguajes con estructuras modulares y conceptos orientados a objetos.

Hay un dilema cierto entre la expresividad de un lenguaje y los niveles de abstracción que soporta. Hacer más expresivo un lenguaje facilita especificaciones más breves y elegantes, pero también puede hacer más difícil el razonamiento

***Mito 14.*** *La gente de métodos formales siempre usa métodos formales.*

Existe la creencia generalizada de que los proponentes de los métodos formales los aplican en todos los aspectos del desarrollo de sistemas. Esto no podría estar más lejos de la realidad. Incluso los más fervientes defensores de los métodos formales reconocen que algunas veces los otros enfoques son mejores.

En el diseño de las interfaces de usuario, por ejemplo, es muy difícil para el desa-

rrollador determinar, y por lo tanto formalizar, los requerimientos exactos de interacción humano-computadora al comienzo del proyecto. En muchos casos, la interfaz de usuario debe ser configurable, con varias combinaciones de colores subrayando ciertas condiciones (como que el rojo denota una situación indeseable). La dificultad mayor, sin embargo, es determinar cómo se va a ver y sentir la interfaz. Si una interfaz es apropiada o no es subjetivo, no sensible a una investigación formal. Aunque se han hecho varios intentos (algunos exitosos) de especificar formalmente las interfaces de usuario, en general una prueba de conformidad aquí cae en el dominio del razonamiento informal.

Hay muchas otras áreas en las cuales, aunque es posible, no es práctico utilizar formalizaciones debido a los recursos, tiempo, o dinero. La mayoría de los proyectos exitosos con métodos formales involucran la aplicación de los métodos formales a porciones críticas del desarrollo del sistema. Claramente, *el desarrollo de sistemas debería ser tan formal como fuese posible, pero no más formal.*

### Diez Mandamientos de los Métodos Formales

Los métodos formales permiten especificaciones más precisas y una detección más temprana de los errores. Bowen y Hinchey [BH95b] recomiendan que los desarrolladores de software que quieran obtener beneficios de los métodos formales deberían prestar atención a los siguientes lineamientos.

1. **Elegir una notación apropiada:** El lenguaje de especificación es la herramienta principal del especificador durante las etapas iniciales del desarrollo del sistema. La notación debería tener una semántica formal bien definida.

2. **Formalizar, pero no formalizar de más:** Aplicar métodos formales a todos los aspectos del sistema es innecesario, además de costoso.

3. **Estimar costos:** El hecho de que muchos proyectos que utilizan métodos formales excedan sus presupuestos no prueba que los métodos son más caros, sino que las técnicas de estimación de costos para estos métodos necesitan mejorar.

4. **Tener un gurú en métodos formales a quien llamar:** La mayoría de los proyectos que utilizan métodos formales se han apoyado en al menos un consultor con experiencia en técnicas formales.

5. **No abandonar los métodos de desarrollo tradicionales:** Sería mejor integrar de manera efectiva los métodos formales en los procesos de diseño existentes. Idealmente, cualquier combinación de métodos estructurados y formales debería resaltar lo mejor de cada uno.

6. **Documentar lo suficiente:** La documentación es importante para el proceso de diseño, particularmente si posteriormente se deberán introducir cambios. Formalizar la documentación reduce tanto los errores como la ambigüedad. En sistemas críticos de seguridad, por ejemplo, documentar cuestiones cronológicas es especialmente importante.

7. **No comprometer los estándares de calidad:** Los métodos formales son un medio para lograr una mejor integridad del sistema *cuando se aplican apropiadamente*. No alcanza con aplicarlos para desarrollar software correcto. Por lo tanto la organización debe continuar satisfaciendo sus estándares de calidad.

8. **No ser dogmático:** Los métodos formales son una opción entre muchas técnicas. No se deberían descartar otros métodos, ya que los métodos formales no pueden garantizar la correctitud.

9. **Probar, probar, y volver a probar:** La fase de pruebas puede demostrar la presencia de errores, pero no su ausencia. Que un sistema haya pasado las pruebas de unidad y de sistema no implica que esté libre de errores.

10. **Reusar:** El reuso puede aplicarse tanto a los métodos de desarrollo formales como a los más convencionales. Teóricamente, el reuso puede contrarrestar algunos costos fijos como herramientas, entrenamiento y educación.

### 4.1.6. Limitaciones de los Métodos Formales

Dada la aplicatividad de los métodos formales a lo largo del ciclo de vida, y sus posibilidades que abarcan casi todas las áreas de la ingeniería de software, ¿por qué no son más visibles? Parte del problema es educativo. Las revoluciones no se hacen mediante conversión, sino por la muerte de la vieja guardia. Los graduados universitarios más recientes tienden a estar más dispuestos a experimentar con métodos formales.

Por otro lado, la única barrera para la transición generalizada de esta tecnología no es la falta de conocimiento por parte de los prácticos. Los métodos formales sufren

de ciertas limitaciones. Algunas de éstas son inherentes y nunca se superarán. Otras restricciones, con investigación y práctica, se eliminarán a medida que los métodos formales hagan su transición hacia un uso más amplio.

### El problema de los requerimientos

Las limitaciones inherentes a los métodos formales se resumen con nitidez en el aforismo tan frecuentemente mencionado, "no se puede ir de lo informal a lo formal por medios formales". En particular, los métodos formales pueden probar que una implementación satisface una especificación formal, pero no que una especificación formal captura la comprensión informal intuitiva que el usuario tiene de un sistema. En otras palabras, los métodos formales se pueden usar para verificar un sistema, pero no para validarlo. La diferencia es que la validación muestra que un producto satisface su misión operativa, mientras que la verificación muestra que cada paso del desarrollo satisface los requerimientos impuestos por los pasos previos.

No se debería subestimar el alcance de esta limitación. Un influyente estudio de campo [CKI88] encontró que los tres problemas más importantes en el desarrollo de software son:

- El escaso conocimiento del dominio de aplicación
- Cambios en y conflictos entre los requerimientos
- Problemas de comunicación y coordinación

Los proyectos exitosos con frecuencia lo fueron debido al rol de uno o dos diseñadores excepcionales. Estos diseñadores tenían una profunda comprensión del dominio de aplicación y pudieron mapear los requerimientos de las aplicaciones a incumbencias de las ciencias de la computación. Estos hallazgos sugieren que la transformación del conocimiento informal de la aplicación a una especificación formal es un área clave en el desarrollo de grandes sistemas.

La evidencia empírica sugiere, sin embargo, que los métodos formales pueden contribuir al problema de la adecuada captura de requerimientos. La disciplina de producir una especificación formal puede resultar en menos errores de especificación. Más aún, los implementadores que no tienen un conocimiento excepcional del área de aplicación cometen menos errores cuando implementan una especificación formal

que cuando se apoyan en un conocimiento difuso de la aplicación. Estos beneficios pueden existir aún cuando la especificación final esté expresada en lenguaje natural y no en uno formal. Una especificación actúa como un "contrato" entre un usuario y un desarrollador. La especificación describe el sistema a entregar. El uso de especificaciones escritas en un lenguaje formal para complementar las descripciones en lenguaje natural puede hacer más preciso este contrato. Finalmente, los desarrolladores de ambientes de programación automatizada, que usan métodos formales, desarrollaron herramientas para capturar en forma interactiva una comprensión informal del usuario y así desarrollar una especificación formal.

Aún así, los métodos formales nunca pueden reemplazar el conocimiento profundo de la aplicación por parte del ingeniero de requerimientos, ya sea a nivel del sistema o a nivel del software. El conocimiento de la aplicación por parte del diseñador excepcional no se limita a una disciplina. Por ejemplo, una aplicación aeronáutica podría requerir conocimiento de control de vuelos, navegación, procesamiento de señales, y contramedidas electrónicas.

### Implementaciones físicas

La segunda gran brecha entre las abstracciones de los métodos formales y la realidad concreta surge de la naturaleza de cualquier computadora real con existencia física. Los métodos formales pueden verificar que una implementación satisface una especificación cuando corre en una máquina abstracta idealizada, pero no cuando corre en cualquier máquina física.

Algunas de las diferencias entre las máquinas idealizadas típicas y las máquinas físicas son necesarias para que las pruebas de correctitud sean legibles. Por ejemplo, se podría asumir que una máquina abstracta tiene memoria infinita, mientras que cualquier máquina real tiene algún límite superior. De forma similar, las máquinas físicas no pueden implementar números reales, como fueron axiomáticamente descriptos por los matemáticos, mientras que las pruebas se construyen en su mayoría asumiendo la existencia de reales matemáticamente precisos. En principio no hay razón por la cual los métodos formales no puedan incorporar estas limitaciones. Las pruebas, sin embargo, serían mucho más confusas y menos elegantes, y estarían limitadas a una máquina particular.

Sin embargo, aquí sí existe una limitación. Las pruebas formales pueden mostrar

con certeza, sujeta a errores de cálculo, que dadas ciertas suposiciones, un programa es una implementación correcta de una especificación. Lo que no se puede mostrar formalmente es que esas suposiciones son descripciones correctas de un sistema físico real. Un compilador puede no implementar correctamente un lenguaje como fue especificado. Por lo que una prueba de un programa en ese lenguaje no podrá garantizar el comportamiento de la ejecución del programa bajo ese compilador. El compilador puede ser verificado formalmente, pero esto sólo desplaza el problema a un nivel más bajo de abstracción. Los chips de memoria y los puertos pueden tener fallas. No importa cuán cuidadosamente se verifique formalmente una aplicación, en algún punto se debe aceptar que un sistema físico real satisface los axiomas usados en una prueba.

Tanto críticos como desarrolladores de métodos formales están muy conscientes de esta limitación, aunque los críticos no siempre parecen estar conscientes de las declaraciones explícitas de los desarrolladores sobre este punto. Esta limitación no significa que los métodos formales sean inútiles. Las pruebas formales aíslan explícitamente aquellos lugares donde puede ocurrir un error. Éstos pueden surgir al proveer una máquina que implemente una máquina abstracta, con la suficiente precisión y eficiencia, sobre la cual se basan las pruebas. Dada esta implementación, una prueba incrementa en gran medida la confianza en un programa.

Aunque ninguno de los prominentes defensores de los métodos formales recomienda evitar completamente las pruebas, no es claro qué rol pueden jugar para aumentar la confianza en las áreas no abordadas por los métodos formales. Las áreas abordadas por las pruebas y los métodos formales pueden superponerse, dependiendo de las metodologías específicas empleadas. Desde un punto de vista abstracto, la pregunta de qué conocimiento o creencia racional se puede proveer mediante las pruebas es la clave de la base racional de la inducción. ¿Cómo puede una observación de que algunos objetos de un tipo dado tienen una cierta propiedad convencer a alguien de que todos los objetos de ese tipo tienen la propiedad? ¿Cómo podría una demostración de que un programa produce las salidas correctas para ciertas entradas conducir a la creencia de que el programa probablemente producirá la salida correcta para todas las entradas? Si un compilador procesa correctamente ciertos programas, como está definido mediante un estándar sintáctico y semántico, ¿por qué se debería concluir que se podría confiar en cualquier axioma semántico en el estándar para una prueba formal de la correctitud de un programa no usado para probar el compilador? Hace más de

dos siglos atrás, el filósofo británico David Hume puso cuestiones relacionadas en el centro de su epistemología.

Dos siglos de debate no alcanzaron para llegar a un consenso sobre la inducción. Los seres humanos todavía se inclinan a sacar estas conclusiones. Los desarrolladores de software muestran la misma inclinación al probar programas de computadora. Los métodos formales nunca reemplazarán enteramente a las pruebas, ni sus defensores pretenden que sea así. En principio, siempre existe una brecha entre la realidad física y lo que puede verificarse formalmente. Con el uso más difundido de los métodos formales, sin embargo, el rol de las pruebas cambiará.

### Cuestiones de implementación

Las brechas entre las intenciones de los usuarios y las especificaciones formales y entre las implementaciones físicas y las pruebas abstractas crean limitaciones inherentes a los métodos formales, no importa cuánto puedan desarrollarse en el futuro. También hay algunos asuntos pragmáticos que reflejan el estado actual de la tecnología.

La introducción de una nueva tecnología en una organización de software a gran escala no es simple, en particular una tecnología tan potencialmente revolucionaria como los métodos formales. Se deben tomar decisiones sobre si la tecnología debería ser adoptada completa o parcialmente. Se deben adquirir herramientas de apoyo apropiadas. Se debe reeducar al personal actual, y puede ser necesario contratar personal nuevo. Se deben modificar, quizás en forma drástica, las prácticas existentes. Todos estos temas surgen con los métodos formales. Las decisiones óptimas dependen de la organización y de las técnicas para implementar los métodos formales. Existen varios esquemas con varios niveles de factibilidad e impacto.

Sin embargo surge la pregunta de si los métodos formales son aplicables en implementación a gran escala. La mayoría de ellos están desarrollados para abordar mejor cuestiones de funcionalidad y seguridad, pero incluso para esos métodos maduros existen serias dudas sobre su habilidad para adaptarse a aplicaciones grandes. En trabajos académicos, una prueba de cien líneas de código puede verse como un logro. La aplicatividad de tales métodos a un sistema comercial o militar, que puede tener más de un millón de líneas de código, está seriamente en duda. Este problema de escalamiento puede ser un factor decisivo en la elección de un método.

Un esquema adoptado con frecuencia para usar métodos formales en proyectos del mundo real es seleccionar un pequeño subconjunto de componentes para tratamiento formal, solucionando así, en parte, el problema de la escalabilidad. Estos componentes deberían seleccionarse bajo criterios de seguridad o criticismo. Los componentes particularmente sensibles a pruebas formales deberían ser seleccionados específicamente. De esta forma, se evita el alto costo de los métodos formales para el proyecto completo, sólo se incurre en ellos donde los requerimientos del proyecto los justifican. Bajo este esquema se evita el problema del escalamiento, ya que los métodos formales nunca se aplican a gran escala.

Las decisiones sobre adquisición e integración de herramientas deben ser cuidadosamente consideradas. Los defensores de los métodos formales argumentan que éstas deberían integrarse al proceso de diseño. No se desarrolla una especificación y una implementación y luego se intenta probar que la implementación satisface la especificación. Más bien, se diseña la implementación y la prueba en paralelo, en continua interacción. A veces la discusión sobre verificadores automáticos sugiere que el primer enfoque, y no el segundo, provee un modelo de implementación. La implementación selectiva de métodos formales en pequeñas porciones de proyectos grandes puede hacer que esta integración sea difícil de lograr.

Otro enfoque puede tener impactos mucho más globales. Quizás se debería descartar por completo el ciclo de vida en cascada. Un enfoque alternativo es desarrollar especificaciones formales al principio del ciclo de vida y luego derivar automáticamente el código fuente para el sistema. El mantenimiento, las mejoras y modificaciones se realizarán sobre las especificaciones, repitiendo luego este proceso de derivación. Los programadores son reemplazados, o al menos fuertemente guiados, por un conjunto inteligente de herramientas integradas. El conocimiento sobre métodos formales está embebido en las herramientas, usando técnicas de Inteligencia Artificial para dirigir el uso de los métodos formales. Esta revolucionaria metodología sin programadores todavía no existe, pero está inspirando a muchos desarrolladores de herramientas.

Una tercera alternativa es introducir parcialmente los métodos formales a lo largo de un proyecto u organización, pero permitiendo un nivel variable de formalidad. En este sentido, la verificación informal es un argumento que sugiere que los detalles deben completarse para proveer una prueba completamente formal. El ejemplo más conocido de esta alternativa es la metodología *Cleanroom*, desarrollada por Harlan

Mills [MDL87]. Dado que los niveles de formalidad son variables, las herramientas son mucho menos útiles bajo este enfoque. La metodología *Cleanroom* implica mucho más que métodos formales, pero éstos están completamente integrados en la metodología. Otras tecnologías involucradas incluyen el ciclo de vida en espiral, el modelado de confiabilidad de software, un enfoque de pruebas específico, certificación de confiabilidad, inspecciones y control de proceso estadístico. Así, aunque este enfoque permite la experimentación parcial con métodos formales, aún requiere cambios drásticos en la mayoría de las organizaciones.

No importa en qué medida una organización decide adoptar métodos formales, si lo hace; siempre surgen cuestiones de entrenamiento y educación. La mayoría de los programadores o no han sido expuestos al trasfondo matemático necesario, o no lo usan en su práctica diaria. Incluso aquellos que comprenden completamente las matemáticas pueden no haberse dado cuenta de su aplicatividad al desarrollo de software. La teoría de conjuntos generalmente se enseña en cursos de matemática pura, no de programación. Incluso la matemática discreta, un curso estándar cuya ubicación en los planes de cátedra universitaria le debe mucho a los ímpetus de las asociaciones de profesionales en ciencias de la computación, muy pocas veces se liga a aplicaciones de software. La educación en métodos formales no debería confinarse a los programas universitarios de grado, para estudiantes que recién ingresan al campo. Se deberían encontrar los medios, como seminarios y cursos de extensión, para reeducar la fuerza de trabajo existente. Quizás este problema educativo sea el mayor obstáculo para una transición más generalizada a los métodos formales.

## 4.2. Lenguajes y Métodos Formales

### 4.2.1. Introdución

Antes de comenzar a examinar en detalle la metodología RAISE, se introducirá una descripción resumida de otros métodos y lenguajes formales existentes.

#### Lenguajes de especificación y de programación

Aquí se examinan lenguajes que se usan para escribir especificaciones y programas en distintos enfoques de desarrollo formal de programas [San88]. En muchos enfoques,

la distinción entre especificaciones y programas es borrosa o incluso inexistente, como lo sugiere la evolución gradual de una especificación de alto nivel hacia un programa.

Algunos enfoques adoptan un único lenguaje, llamado de *amplio espectro*, que puede usarse para escribir especificaciones de alto nivel, programas eficientes, y cualquier cosa que surja en la transición desde los primeros hasta los últimos durante el proceso de desarrollo. En estos pasos intermedios es natural que los términos de la especificación se mezclen libremente con los de la programación, debido a la forma en que las especificaciones de alto nivel se refinan hacia los programas. Esto también evita varios problemas que surgen cuando se usan lenguajes separados para la especificación y la programación: no hay diferencias esenciales entre el refinamiento de programas y el refinamiento de especificaciones; los mismos constructores de modularización se pueden usar para estructurar tanto especificaciones como programas; no hay saltos bruscos de una notación a la otra sino más bien una transición gradual desde la especificación de alto nivel hacia el programa eficiente.

En este contexto, las "especificaciones de alto nivel" son descripciones que dan detalles de *qué* se requiere. Esto contrasta con los "programas", que sugieren *cómo* se va a calcular el resultado deseado. La palabra "especificación" se usará para referir a cualquier descripción del comportamiento de entrada/salida de un sistema, ya sea algorítmico o no; así, un programa es una especificación ejecutable. Esto es consistente con la terminología usada en los lenguajes de amplio espectro, donde un programa es una especificación que sólo usa el subconjunto ejecutable del lenguaje.

En algunos enfoques se asegura que la especificación inicial de alto nivel de los requerimientos debe ser ejecutable. Se piensa que esto es necesario para asegurar que esta especificación formal refleje fielmente las intenciones del cliente; con una especificación ejecutable esto puede verificarse mediante las pruebas. Sin embargo, requerir que la especificación inicial sea ejecutable significa que una parte importante del proceso de desarrollo del programa no se formalice. Para construir una especificación ejecutable es necesario tomar muchas decisiones que podrían dejarse abiertas en una no ejecutable. Esto significa que innecesariamente se elimina desde el comienzo la consideración de toda una variedad de implementaciones alternativas perfectamente aceptables. Apuntar a una especificación inicial que sea lo más abstracta y no algorítmica posible con frecuencia conduce a simplificaciones y generalizaciones útiles que de otra manera no se hubiesen descubierto.

### *VDM*

VDM (Vienna Development Method), presentado en [Jon80] y [BJ82] es un método para desarrollo *riguroso* (no formal) de programas. El objetivo es producir programas mediante un proceso en el cual se demuestra que los pasos individuales de refinamiento son correctos usando argumentos formalizables antes que formales, aproximando así el nivel de rigor usado en matemática. Se supone que esto logra la mayoría de las ventajas del desarrollo formal de programas asegurando que se evita todo descuido pero sin la recarga fundacional y notacional de la formalidad pura.

VDM es uno de los enfoques de desarrollo sistemático de programas más ampliamente aceptados. Para describir tipos de datos, utiliza una estrategia orientada al modelo. Los modelos se construyen usando funciones, relaciones y conjuntos.

Un problema con las especificaciones orientadas al modelo es que es fácil especificar por demás un sistema, eliminando desde el principio la consideración de ciertas implementaciones. En VDM se ha estudiado una noción precisa de sobre-especificación. Se dice que un modelo es *sesgado* si no es posible definir una prueba de igualdad sobre los valores de los datos del modelo en términos de los operadores definidos. Intuitivamente, un modelo sesgado contiene redundancia innecesaria. Un modelo insesgado se ve como lo suficientemente abstracto como para ser la especificación inicial de alto nivel de un sistema. Durante el proceso de refinamiento se introducen modelos más concretos.

Para especificar procedimientos, que pueden tener efectos colaterales, se utilizan pre y poscondiciones. La descomposición durante el refinamiento es una forma de dividir procedimientos en sentencias individuales que pueden a su vez ser especificadas usando pre y poscondiciones. Cuando este proceso está completo, el resultado es un programa.

### *Z*

Z es un lenguaje de especificación que se basa en el principio de que los programas y datos pueden describirse usando teoría de conjuntos. Así, Z no es más que una notación formal para la teoría común de conjuntos. La primera versión de Z [ASM80] usaba una notación más bien torpe y poco concisa, pero las versiones posteriores [Spi88, Spi89] adoptaron una notación más elegante y concisa, basada en la idea de *esquemas*.

Los tipos de datos se modelan en Z usando construcciones de la teoría de conjun-

tos, así como los "tipos de datos" matemáticos, como número naturales, reales, pares ordenados y secuencias se definen, en matemáticas, en términos de la teoría de conjuntos. Como en el caso de VDM, en Z también pueden usarse pre y poscondiciones para especificar procedimientos con efectos colaterales.

***Lenguajes basados en reglas de reescritura***

En la década de los 80's se desarrollaron algunos lenguajes de programación de muy alto nivel, que pueden verse como lenguajes de especificación ejecutables. Se basan en la idea de que las ecuaciones pueden verse como reglas de reescritura. Es decir, una ecuación $\forall X.t = t'$ puede verse como una regla de reescritura $t \Rightarrow t'$ (o $t' \Rightarrow t$) que dice que cualquier instancia de $t$ en una expresión puede ser reemplazada por la correspondiente instancia de $t'$. Bajo ciertas condiciones, es posible "ejecutar" un conjunto de esas reglas para calcular el valor de una expresión.

Estos lenguajes tienen relación con los lenguajes de programación lógica, que también han sido anunciados como un lenguaje de especificación ejecutable. En [GM86] puede verse la relación entre la programación basada en reglas de reescritura y la programación lógica.

Algunos de los lenguajes de este tipo son:

**HOPE** [BMS80]: lenguaje de programación puramente aplicativo, con un sistema de tipos rico pero seguro, que permite al usuario definir nuevos tipos de datos. Requiere que las ecuaciones sean de la forma $f(patron) \Leftarrow expresion$, donde *patron* es una expresión que sólo contiene variables y constructores. Esta restricción sintáctica es lo que hace que los programas HOPE sean ejecutables. También se permiten funciones de más alto orden, que toman funciones como argumentos y/o devuelven funciones como resultado. Además tiene otras características como facilidades de modularización de programas y *tipos polimórficos* de estilo Milner [Mil78].

**ML Estándar** [Har86, HMM86]: en su esencia es similar a HOPE, aunque tiene importantes avances, como sus poderosas habilidades para la modularización de programas, provistas por la definición separada de interfaces (*signaturas*) y sus implementaciones (*estructuras*). Cada estructura tiene una signatura que da los nombres de los tipos y funciones definidos en la estructura. Las estructuras pueden construirse por sobre estructuras ya existentes, por lo que en realidad cada

una es una *jerarquía* de estructuras, y esto también se refleja en su signatura. Además, ML Estándar tiene muchas otras características, como un poderoso mecanismo de manejo de excepciones, tipos polimórficos y algunos constructores imperativos.

**OBJ2** [FG+85]: la motivación inicial de este trabajo fue permitir la prueba de especificaciones algebraicas, aunque ahora se advierte como un lenguaje de programación de ultra alto nivel. Un programa OBJ2 (llamado *objeto*) declara algunos tipos y funciones nuevos que luego se definen por medio de un conjunto de ecuaciones. Cuando se ven como reglas de reescritura, se requiere que las ecuaciones tengan las propiedades de Church-Rosser y de terminación, lo que garantiza que las repetidas reescrituras usando las reglas siempre terminarán con un resultado único. Esto permite ecuaciones no admitidas en HOPE ni en ML Estándar, y no requiere que se distinga entre constructores y otras funciones. Otras características de OBJ2 son una sintaxis muy flexible y una noción de subtipo. OBJ2 no permite funciones de mayor orden, en contraste con HOPE y ML Estándar.

***Lenguajes de especificación algebraica***

Se ha dedicado mucho trabajo a métodos de especificación basados en la idea de que un programa funcional puede modelarse como un número de conjuntos de valores de datos (un conjunto de valores para cada tipo de datos) junto con un número de funciones (totales) sobre esos conjuntos, correspondientes a las funciones del programa. Esto es una abstracción de los algoritmos usados para calcular las funciones y de cómo se expresan esos algoritmos en un determinado lenguaje de programación, y permite concentrarse en la representación de los datos y en el comportamiento de entrada/salida de las funciones. Es posible extender este paradigma para manejar también programas imperativos, modelándolos como programas funcionales o usando una noción algebraica diferente. La motivación inicial de este trabajo fue proveer una base formal para el uso de abstracción de datos en el desarrollo de programas.

Los primeros trabajos en esta área fueron [Zil74], [Gut75] y [GTW78], de los cuales el último es el más formal. Una especificación consiste en una *signatura* –un conjunto de *tipos* (nombres de tipos de datos) y un conjunto de nombres de funciones con sus tipos– más un conjunto de axiomas que expresan restricciones que las funciones deben satisfacer. A continuación se describen algunos lenguajes de esta categoría.

**CLEAR** : este lenguaje de especificación [BG80, BG81, San84] provee un pequeño número de operaciones que permiten la construcción de especificaciones grandes y complejas de forma estructurada partiendo de piezas pequeñas, comprensibles y reusables. Las operaciones brindan formas de combinar dos especificaciones, de enriquecer una especificación con nuevos tipos, funciones y axiomas, de renombrar y/o eliminar algunos de los tipos y funciones de una especificación y de construir y aplicar especificaciones *parametrizadas*. Existe la posibilidad de dejar algunas decisiones deliberadamente abiertas, para que sean tomadas luego en el proceso de desarrollo del programa.

**Larch** : es una familia de lenguajes de especificación [GH83, GH93] desarrollada para dar soporte al uso productivo de las especificaciones formales en la programación. Uno de sus objetivos es respaldar varios lenguajes de programación diferentes, incluyendo lenguajes imperativos. Cada lenguaje Larch se compone de dos partes: el *lenguaje de interfaz*, que es específico al lenguaje particular que se está considerando, y el *lenguaje compartido*, que es común a todos los lenguajes. El primero se usa para especificar módulos de programa mediante lógica de predicados con igualdad y constructores para afrontar efectos colaterales, manejo de excepciones y otros aspectos del lenguaje dado; y el segundo es un lenguaje de especificación algebraica usado para describir abstracciones independientes del lenguaje de programación.

**ML Extendido** [ST86]: es un lenguaje de amplio espectro obtenido de la extensión de ML Estándar para permitir que los axiomas aparezcan en las signaturas y en lugar del código en definiciones de estructuras y funtores. Los axiomas en las signaturas restringen el comportamiento permitido de los componentes de las estructuras que se corresponden con esa signatura. Los axiomas en una estructura o funtor se usan para definir funciones y datos en una forma de alto nivel que no es necesariamente ejecutable.

### Refinamiento de especificaciones

Resulta sorprendente la dificultad que existe para dar una definición precisa que capture la noción intuitivamente simple de refinamiento. El problema principal es el de la representación de los datos. Durante el proceso de refinar una especificación abs-

tracta para lograr un programa concreto, es necesario idear representaciones de datos cada vez más concretas. En última instancia, todos los datos deben ser representados usando los tipos de datos primitivos provistos por el lenguaje de programación elegido.

Cualquiera sea la noción formal de refinamiento que se adopte, es esencial para la correctitud del método de desarrollo que los pasos de refinamiento puedan componerse de dos maneras. Primero, dos pasos de refinamiento $SP \rightarrow SP'$ y $SP' \rightarrow SP''$ se deberían poder componer para dar un refinamiento correcto $SP \rightarrow SP''$, para especificaciones o programas arbitrarios $SP$, $SP'$ y $SP''$. Esta es la propiedad de los pasos de refinamiento (conocida como *composición vertical* [GB80]) que garantiza la correctitud de los programas desarrollados a partir de especificaciones en una modalidad por pasos. Segundo, si la estrategia de desarrollo del programa permite que las especificaciones se descompongan en unidades más pequeñas durante el proceso de desarrollo, entonces la noción de refinamiento adoptada debe ser compatible con las operaciones de construcción de especificaciones: dados dos pasos de refinamiento $SP_1 \rightarrow SP_1'$ y $SP_2 \rightarrow SP_2'$, debería ocurrir que $SP_1 \oplus SP_2 \rightarrow SP_1' \oplus SP_2'$ sea un refinamiento correcto para cualquier operación de construcción de especificaciones $\oplus$. Esta es la propiedad (conocida como *composición horizontal*) que garantiza que hilos de desarrollo separados puedan proseguir de forma independiente y luego combinarse para lograr un resultado correcto. Finalmente, un método de desarrollo formal debe proveer alguna forma de probar que los pasos de refinamiento son correctos con respecto a la noción de refinamiento adoptada.

***VDM***

En VDM, que ya se introdujo en la Sección 4.2.1, cuando se desea establecer que una especificación $SP'$ es un refinamiento correcto de otra $SP$, la prueba de correctitud consiste en los siguientes pasos:

- Definir una "función de recuperación" *recup* que mapea los valores de los datos especificados en $SP'$. Ésta relaciona valores de datos concretos con los valores abstractos que representan. Debido a la dirección de *recup*, puede haber muchos valores concretos que representan un único valor abstracto, pero no a la inversa.

- Probar que *recup* es total (sobre los valores de datos en $SP'$ que satisfacen el invariante del tipo de datos) y suryectiva. Esto garantiza que todos los va-

lores de datos concretos representen algún valor abstracto, y que todo valor abstracto tenga una representación. El invariante del tipo de datos en $SP'$ debería restringir el dominio de *recup* a los valores concretos que se usarán como representaciones.

- Identificar una operación $f'$ en $SP'$ correspondiente a cada operación $f$ en $SP$. Se supone que las operaciones en el nivel concreto modelan las del nivel abstracto, como se asegurará en los últimos dos pasos.

- Probar que pre–$f(recup(v)) \Rightarrow$pre–$f'(v)$ para todo valor concreto $v$. Esto asegura que las precondiciones de las operaciones concretas no son más restrictivas que las de las operaciones abstractas correspondientes.

- Probar que pre–$f(recup(\overline{v}))\wedge$post–$f'(\overline{v}, v) \Rightarrow$post–$f(recup(\overline{v}), recup(v))$ para todo valor concreto $\overline{v}, v$. Esto garantiza que los resultados producidos por las operaciones concretas reflejen las producidas por las operaciones abstractas.

Una ventaja de esta estrategia de VDM es que los pasos para verificar la correctitud son muy explícitos y relativamente fáciles de ejecutar. Esto no siempre se verifica en los demás enfoques.

***Z***

La estrategia de Z (que ya se introdujo en la Sección 4.2.1) para el refinamiento de especificaciones que se describe en [Spi89] es virtualmente idéntica a la estrategia de VDM presentada anteriormente, excepto por diferencias en su notación y terminología, lo que lleva a las mismas ventajas y desventajas. Allí también se asegura que es posible manejar situaciones en las que varios valores abstractos se representan con un único valor concreto (como a veces lo requiere una especificación sesgada) usando una estrategia ligeramente más complicada.

***Enfoques algebraicos***

La mayor parte del trabajo en este contexto sobre refinamiento de especificaciones se ha inspirado en el trabajo de Hoare [Hoa72] sobre refinamiento de datos. Para Hoare, un álgebra $A'$ es un refinamiento de un álgebra $A$ si existe alguna subálgebra $A''$ de $A$ con un homomorfismo suryectivo $h : A'' \rightarrow A$. Hay una íntima conexión con el enfoque VDM descrito anteriormente: la subálgebra $A''$ contiene aquellos valores de datos que satisfacen el invariante del tipo de datos (el *invariante de representación* en

la terminología de Hoare) y $h$ es la función de recuperación (*función de abstracción*). Requerir que $h$ sea un homomorfismo garantiza que las operaciones de $A''$ (y por lo tanto las de $A'$) se comporten igual que las operaciones de $A$.

Esta idea se puede extender del álgebra a las especificaciones si se dice que una especificación $SP'$ es un refinamiento correcto de otra especificación $SP$ si cada modelo de $SP'$ refina un modelo de $SP$ en el sentido de Hoare. Es posible modificar esta definición en varios sentidos, para tener en cuenta la posibilidad de que $SP'$ contenga operaciones con nombre diferentes de los de $SP$, y para permitir la construcción de una especificación intermedia en base a $SP'$.

### 4.2.2. RAISE: Introducción y Características

Luego de haber presentado un pantallazo general sobre algunos de los métodos y lenguajes formales existentes, se concentrará la atención en el método RAISE. RAISE es el acrónimo de "Enfoque Riguroso a la Ingeniería de Software Industrial" (**R**igorous **A**pproach to **I**ndustrial **S**oftware **E**ngineering). Le da su nombre a un lenguaje de especificación formal, el Lenguaje de Especificación RAISE (RSL) [G+92], un método asociado y un conjunto de herramientas. RSL es un lenguaje de amplio espectro: se puede usar para formular tanto las especificaciones iniciales, muy abstractas, y para expresar diseño de bajo nivel, adecuado para su traducción a lenguajes de programación. Por lo tanto el método abarca:

- la formulación de especificaciones abstractas
- el desarrollo de éstas hacia especificaciones sucesivamente más concretas
- la justificación de la correctitud del desarrollo
- la traducción de la especificación final a un lenguaje de programación

El método RAISE se basa en cuatro principios:

- Desarrollo separado
- Desarrollo por etapas
- Inventar y verificar
- Rigor

### Desarrollo separado

Si se quiere desarrollar sistemas de cualquier tamaño, hay que tener la capacidad de descomponer su descripción en componentes y componer el sistema desde los componentes desarrollados. También es claro que para la mayoría de los sistemas es necesario tener personas trabajando en distintos componentes al mismo tiempo. Por lo tanto habrá entidades –archivos, documentos, etc.– que se estarán compartiendo. Esto genera dos dificultades: la primera es que debe estar claro quién es el responsable de actualizar tales entidades compartidas, y cuál es el estado de cada versión en cada momento particular; la segunda es que no debe haber ambigüedad acerca de qué significan tales entidades compartidas, y esto causa aún más problemas. Es bastante fácil compartir información sobre el nombre de una función, qué parámetros tiene, o qué tipo tiene su resultado. Pero no es tan fácil ser exactos sobre la semántica de la función.

Si también se considera el desarrollo de estos componentes compartidos, se descubre un problema adicional: ¿qué puede hacer el desarrollador, en forma segura, y que no afecte a los usuarios?

Lo que se necesita es una declaración clara y sin ambigüedad que actúe como un acuerdo o *contrato* entre el desarrollador y los usuarios. Para el desarrollador, un contrato dice qué es lo que debe proveer; para el usuario, le dice qué puede asumir. Esto también significa que se debe conocer quiénes son los usuarios y desarrolladores, de forma que esté claro quiénes van a estar involucrados en una renegociación.

Una especificación de un módulo (o quizás un grupo de módulos) puede actuar como este contrato. Una especificación dice precisamente cuáles son las propiedades esenciales de lo que se está especificando. Una especificación permite una precisión controlable; puede ser tan precisa o imprecisa como los especificadores y usuarios lo requieran.

La Figura 4.1 muestra cómo trabaja el desarrollo separado en el caso simple del desarrollo de un módulo $A$ que se usa en un módulo $B$. Las versiones iniciales de $B$ y $A$ son $B_0$ y $A_0$ y éstas se desarrollan en $n$ y $m$ pasos hasta $B_n$ y $A_m$ respectivamente. Cuando el desarrollo está completo se efectúa la integración usando $A_m$ en lugar de $A_0$ en $B_n$, para formar $B_{n+1}$.

Lo que se quiere es que el sistema final ($A_m$ y $B_{n+1}$) cumpla con los requerimientos originales. Un conjunto de condiciones suficientes para esto es:

- Los módulos iniciales ($A_0$ y $B_0$) juntos cumplen con los requerimientos, es decir que tienen todas las propiedades requeridas.
- Cada paso de desarrollo de $A$ y $B$ es un paso de implementación, es decir que cada módulo implementa el inmediatamente anterior.

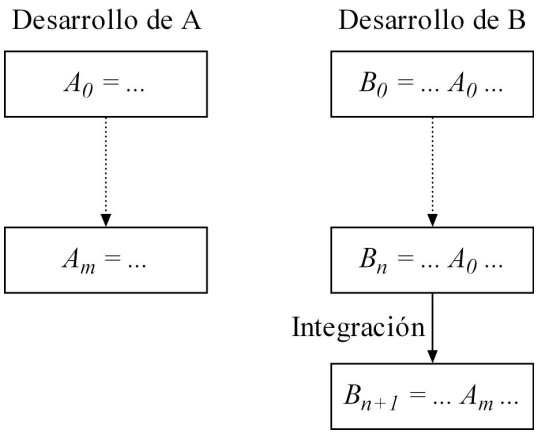


Figura 4.1: Desarrollo separado

Esta visión del desarrollo separado es idealizada. En la práctica esto raramente se sigue precisamente así. Algunas variaciones importantes son:

- No todos los requerimientos se cumplen.
- Los contratos pueden necesitar cambios.
- Algunos pasos de desarrollo pueden no ser implementaciones.

A continuación se analizará cada una de estas variaciones.

***Requerimientos no cumplidos***

Las especificaciones iniciales pueden no cumplir con todos los requerimientos. Esto puede ocurrir por dos razones:

- Algunos requerimientos no se pueden capturar en RSL, porque están fuera de su alcance. Los requerimientos típicos de este tipo pueden ser muy simples, como "el sistema correrá sobre un hardware ... bajo el sistema operativo ... y será codificado en ...", o más complejos, como restricciones de tiempo. Éstos suelen denominarse *requerimientos no funcionales*.

- Algunos requerimientos que pueden capturarse en RSL son diferidos a propósito para más tarde en el desarrollo, debido a que su inclusión al principio complicaría demasiado la especificación inicial.

En cualquiera de los dos casos se deben registrar los requerimientos no cumplidos y asegurarse de que se traten más tarde.

***Contratos cambiantes***

Durante el desarrollo, puede ocurrir que se necesite cambiar un contrato (como $A_0$). Esto puede ser porque los desarrolladores de $B$ necesitan propiedades adicionales a, o incluso distintas de, aquellas por las cuales contrataron originalmente. Puede ser debido a que los desarrolladores de $A$ encuentran imposible o demasiado costoso proveer lo que prometieron en el contrato. En cualquier caso el contrato puede ser renegociado. Este cambio de contratos de $A_0$ y $A_i$ se ilustra en la Figura 4.2.

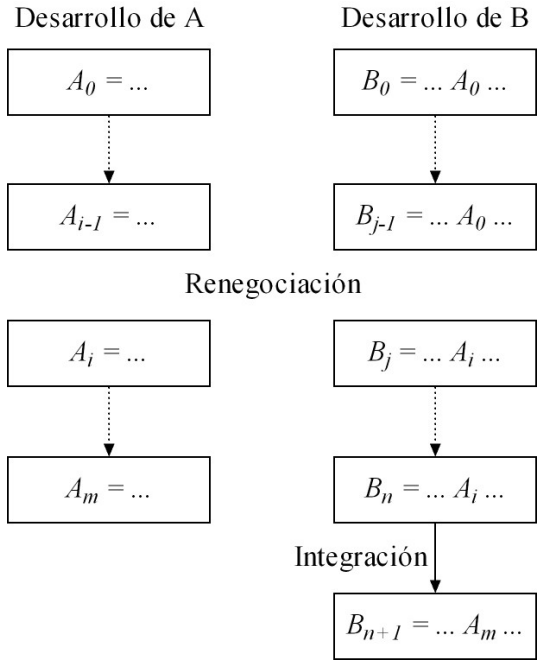


Figura 4.2: Desarrollo separado con contrato cambiado

Esto genera la cuestión de la relación entre los desarrollos antes y después del cambio en el contrato. Puede ser que sencillamente se considere el nivel en el cual tuvo lugar el cambio como un nuevo comienzo. Sin embargo, usualmente, es posible expresar (formalmente) la relación entre $A_i$ y su predecesor, y entre $B_j$ y su predecesor, de forma que se conoce la relación entre el trabajo previo y el nuevo.

***Pasos que no son de implementación***

Puede ocurrir que durante el desarrollo sea necesario realizar pasos que no sean de implementación. Típicamente, el diseño que se quiere elegir sólo funciona para un caso más restringido que aquel con el que se está tratando, y por lo tanto se quiere volver atrás a la especificación inicial para cambiar sus propiedades. El problema es que esto puede resultar en mucho trabajo para un cambio pequeño. Por lo tanto surge la pregunta de si existe algún atajo para situaciones donde se tiene la razonable certeza de que el cambio es menor. Cualquier atajo de este tipo es muy peligroso, pero se pueden examinar algunas posibilidades:

- Si el cambio ocurre en el desarrollo de $A$, todavía puede ser el caso de que el cambio que se quiere hacer es una implementación del contrato. Si se puede establecer esto, no hay problema. Por ejemplo, la forma común de establecer que $A_2$ es una implementación de $A_0$ es que $A_2$ implementa $A_1$ y $A_1$ implementa $A_0$. Pero es completamente válido establecer que $A_2$ implementa $A_0$ directamente. Sin embargo, si el cambio significa que no se está implementando $A_0$, se debería renegociar el contrato. Es muy peligroso debilitar la relación formal entre los desarrollos separados.

- Si el cambio ocurre en el desarrollo de $B$ entonces, de nuevo, se podría estar implementando $B_0$, y no hay un problema real. Pero en general este no es el caso. Al nivel de sistema, sin embargo, no hay otros desarrollos separados y por lo tanto se puede decidir aceptar el cambio a este nivel sin volver sobre los anteriores.

**Desarrollo por etapas**

La discusión sobre desarrollo separado asumía que podría haber varias etapas en el desarrollo de $A$ y $B$. Entonces se puede comenzar con una abstracción adecuada, decidir cuáles son las principales decisiones de diseño que es necesario tomar, y qué dependencias hay entre ellas, y hacer un plan del orden en el cual se abordan. Las decisiones de diseño típicas incluyen:

- Proveer definiciones explícitas para valores a los cuales previamente sólo se les dio signaturas o definiciones implícitas o axiomas.

- Proveer definiciones explícitas para variables y canales a los cuales previamente sólo se refería como **any** (cualquiera).
- Dar definiciones concretas para tipos abstractos.
- Cambiar las definiciones de tipos para permitir funciones más detalladas o (potencialmente) más eficientes sobre ellos (por ejemplo, listas para conjuntos).
- Agregar nuevas definiciones o axiomas.
- Agregar variables de estado, ya sea localmente para almacenar valores o globalmente para reemplazar parámetros.
- Cambiar el estilo de la especificación entre aplicativo e imperativo o entre secuencial y concurrente.
- Agregar parámetros o canales extra para expresar mayor funcionalidad.
- Hacer las cosas más generalmente aplicables (y por lo tanto más reusables) – como agregar parámetros a esquemas o ampliar los tipos de parámetros de las funciones.
- Borrar constructores difíciles de traducir al lenguaje elegido.

Tratar con una o más de tales decisiones significa que se hace un *paso de desarrollo*, se produce una nueva especificación que se puede verificar que conforma con la anterior. Decidir cuáles son las decisiones principales también brinda un plan general de actividades para el desarrollo.

Por supuesto, hay cierta interrelación entre el proceso de diseño y los problemas a abordar. Elegir distintos problemas primero podría resultar en el surgimiento de diferentes problemas. Pero tomar decisiones en esta especie de plan de alto nivel es un paso crítico para comenzar un desarrollo.

El número de etapas de desarrollo desde una especificación inicial puede variar, pero la experiencia sugiere que típicamente es uno o dos. Los componentes principales tenderán a tener más etapas que los sub-componentes.

### Inventar y verificar

Hay técnicas de desarrollo que se apoyan en la transformación. El desarrollador comienza con una expresión y aplica una regla de transformación que crea una expresión distinta pero equivalente. De hecho, hacer una justificación RAISE es un ejemplo de técnica de transformación, pero aplicada a expresiones lógicas en lugar de expresiones de programas.

"Inventar y verificar", por el contrario, es un estilo que permite (en realidad, fuerza) al desarrollador inventar un diseño nuevo. Luego, acto seguido, el desarrollador verifica su correctitud.

La ruta transformacional parece más fácil, porque sólo hay un paso y la correctitud está garantizada. ¿Cuáles son las ventajas de inventar y verificar?

- Los sistemas transformacionales son muy grandes para lenguajes de tamaño razonable.
- En la práctica, las reglas transformacionales sólo se aplican en circunstancias particulares.
- Dado el enfoque inventar y verificar, es fácil inventar varias etapas, quizás cambiando varias veces de idea, antes de decidir sobre el enfoque correcto y proceder a la verificación. Con las transformaciones el enfoque tiende a ser más formal desde el principio.
- En la práctica, algunas etapas de desarrollo no serán implementaciones y pueden, ciertamente, tener poca relación formal con el nivel anterior. Por lo tanto, es necesario hacer pasos "incorrectos", que van en contra de un enfoque que se garantiza correcto.

En principio, éstas no son objeciones al enfoque transformacional. Por supuesto, todas estas observaciones podrían ser hechas en contra de hacer justificaciones simplemente eligiendo reglas y usándolas. Los sistemas transformacionales aún no son los suficientemente poderosos para hacer transformacional todo el desarrollo para un lenguaje como RSL.

**Rigor**

El desarrollo es difícil; las pruebas son aún mucho más difíciles. No es práctico probar todo. A menudo se aprende mucho de la falla de una prueba, pero se aprende mucho más de sus detalles precisos. Por lo tanto, no es suficiente sólo la capacidad de realizar una prueba, sino que es necesaria la capacidad de *explorar* las propiedades de las especificaciones. Otro problema es que, así como hay demasiados casos de prueba para ejecutar en tiempo real, también hay demasiadas propiedades posibles.

Por supuesto, existe el peligro de que, al seleccionar los casos para probar, se pierdan los errores porque es "obvio" que ellos no están ahí. Por lo tanto, puede ser necesario seleccionar las propiedades dignas de ser investigadas más de cerca, y probar formalmente sólo aquellas que se sospecha. A un argumento que puede ser completa o parcialmente informal se le llama *justificación*. Los argumentos que contienen etapas informales se llaman *rigurosos*. Una justificación que es completamente formal es una *prueba*.

### 4.2.3. El rol de RAISE en la ingeniería de software

Un método es un medio para lograr el desarrollo de un *sistema de software*. Por "sistema de software" se entiende:

- un programa, o conjunto de programas conectados, escritos en algún lenguaje ejecutable.
- documentación asociada que sustenta el uso y mantenimiento del programa o conjunto de programas.

Un método consiste esencialmente en procedimientos a seguir y técnicas que facilitan los procedimientos. Los procedimientos o actividades en general se describen en varias capas. A la cabeza están las actividades principales que colectivamente forman el ciclo de vida del software. Dentro de estas actividades principales existen actividades componentes, como producir un tipo particular de documento, seguir un procedimiento de control de calidad, cambiar un documento existente, etc.

Junto con estos procedimientos que identifican varios tipos de actividades, hay técnicas particulares que pueden usarse. La mayor parte del método RAISE consiste en técnicas para cuatro procedimientos principales:

- Especificación
- Desarrollo
- Justificación
- Traducción

Las técnicas tienen que estar relacionadas con los procedimientos que las requieren. Los proveedores y compradores particulares de software usualmente tienen su propia noción de qué método se va a usar, por lo general embebido en sus estándares de calidad. Tales estándares describirán en detalle qué se entiende por procedimientos como "especificación de pruebas", por ejemplo. El uso de métodos formales afectará a algunos de éstos mucho más que a otros. Por lo tanto, no se quiere proponer un método para reemplazar todo lo que ya está en su lugar. Más bien se quiere explicar cómo afectará el método RAISE a los métodos típicos ya en uso.

Se usarán los siguientes términos que describen los procedimientos de nivel superior que se ven más afectados por el uso de RAISE:

**Especificación:** comienza con los requerimientos identificados, escritos en su mayor parte en lenguaje natural, y produce una descripción en RSL. La especificación definirá el comportamiento del sistema en suficiente detalle como para alcanzar todos los requerimientos funcionales principales.

Con frecuencia, a la salida principal de este procedimiento se la denomina *especificación inicial*, porque es la base para especificaciones más detalladas producidas durante el desarrollo. La especificación inicial debería definir *qué* va a hacer el sistema y no *cómo* lo va a hacer. Por lo tanto, lo ideal es que no refleje ninguna decisión de diseño de la arquitectura. En la práctica puede haber alguna variación de esto, por ejemplo la especificación inicial suele incluir algo de diseño arquitectónico.

**Desarrollo:** comienza con la especificación inicial y produce una especificación RSL nueva, más detallada, que conforma con la original y está lista para la traducción, la *especificación final*. Este procedimiento suele llamarse diseño detallado.

**Traducción:** comienza con la especificación final en RSL y produce un programa o conjunto de programas en algún lenguaje ejecutable. Este procedimiento también se llama codificación.

Todos estos procedimientos pueden aplicarse al mantenimiento (cuando las entradas también incluyen el resultado de trabajo previo) como también al desarrollo original.

### Validación y verificación

Hay dos aspectos para mostrar correctitud:

**Validación:** es comprobar que se está creando lo que se requiere, es decir que se están cumpliendo los requerimientos. Se puede expresar como "resolver el problema correcto". La validación es necesariamente informal, porque se está contrastando contra requerimientos escritos en lenguaje natural.

**Verificación:** es comprobar que el proceso de desarrollo es correcto. Incluye la formulación y justificación de relaciones formales entre los pasos del desarrollo. Se expresa como "resolver correctamente el problema". La verificación se puede hacer con distintos grados de formalidad.

### Análisis de requerimientos

La gran fortaleza de los métodos formales es que, al tener que crear una descripción formal de un sistema en una etapa temprana de su desarrollo, uno se ve forzado a tomar decisiones sobre todos los asuntos sobre los cuales los requerimientos no dicen nada o están abiertos a interpretación. Segundo, es particularmente valioso para encontrar errores y resolver ambigüedades o contradicciones en esta etapa temprana. Tercero, la experiencia de los equipos que usan métodos formales dice que su comprensión común del problema mejora al tener que crear y discutir la especificación.

A veces vale la pena rescribir los requerimientos desde la especificación, ya que esto típicamente produce un documento que tiene menos omisiones, contradicciones y repeticiones, está mejor estructurado y es más consistente en su terminología. Una buena especificación inicial es muy importante para que el desarrollo sea rápido y confiable. Por eso es normal que el procedimiento de especificación atraviese varias iteraciones. Ciertamente se espera que, al usar especificación formal y más aún con un desarrollo riguroso, la codificación comience tarde.

### Mantenimiento de la correctitud

Lograr un punto de partida firme y correcto para el diseño detallado es esencial, y es algo que la claridad y lógica de los métodos formales hace posible. Luego se trata de mantener la correctitud; aquí son importantes la relación de implementación en particular y la habilidad general de razonar sobre las especificaciones.

### Concentración en descubrir errores cuando son introducidos

Los métodos pueden estar dirigidos de manera útil a evitar la introducción de errores o a encontrarlos inmediatamente.

Los errores caen en varias categorías:

- Requerimientos que no reflejan los deseos o necesidades reales del cliente o usuario.
- Falla en alcanzar requerimientos como desempeño o confiabilidad, que se trata de diseñar pero que raramente se pueden garantizar de antemano.
- Mala comprensión de los requerimientos o niveles de desarrollo previos.
- Deslices de los desarrolladores o encargados del mantenimiento, al no escribir lo que pretendían escribir.

El uso de métodos formales ayuda con la primera categoría de errores en la medida en que los requerimientos son obviamente incompletos o poco claros. Si éste es el problema, puede ser una buena idea construir un prototipo temprano para mostrar.

Los métodos formales tienen alguna dificultad con la segunda categoría de errores. Los requerimientos de tiempo real son un ejemplo particular, y cómo tratar con ellos es un área de investigación activa. Lo que se puede hacer con tales requerimientos es usarlos como guía de la dirección en la que hay que desarrollar y de los estándares que hay que adoptar.

Los métodos formales también ayudan con la tercera categoría de errores, ya que la formalidad reduce sensiblemente la posibilidad de malos entendidos.

No es claro si las personas son más o menos propensas a errores al escribir RSL de lo que lo serían con otras técnicas. Las herramientas pueden ayudar al hacer notar algunos errores sencillos.

Sin duda, los errores se cometerán de esta forma. Pero las especificaciones formales tienen dos ventajas sobre los programas. Primero, que pueden verificarse contra el nivel anterior. Este proceso es muy efectivo para descubrir equivocaciones. Segundo, las especificaciones formales pueden revisarse. La revisión RSL, para personas expertas en éste y con listas de comprobación de los problemas típicos, es muy efectiva.

#### Producción de documentación que posibilita el mantenimiento

Mantener software es muy caro y propenso a errores. La documentación de diseño típicamente está desactualizada o directamente no existe.

Las especificaciones formales proveen un medio para comprender el código *top-down*. Los comentarios en las especificaciones pueden facilitar este proceso. El proceso también es mucho más fácil (y es más probable que se siga) si toda o la mayor parte de la especificación final es traducible automáticamente.

### 4.2.4. Uso selectivo

Hay dos importantes maneras en las que se puede ser selectivo en el uso de RAISE: en cuánta formalidad se elige tener y a qué componentes de un sistema se le aplica esa formalidad.

#### Grados de Formalidad

El grado de formalidad que se aplica tiene que ser el apropiado al problema y una forma eficiente de atacarlo. Se puede encontrar tres grandes estilos de desarrollo:

**Sólo especificación formal:** la formalidad se aplica al procedimiento de especificación.

**Especificación formal y desarrollo riguroso:** la formalidad se aplica al procedimiento de especificación como antes, pero también al proceso de desarrollo. Esto significa que se escribe tanto las especificaciones abstractas como las más concretas, y también se registran las relaciones de desarrollo entre ellas. Estas relaciones están sujetas a examen y quizás a revisión, pero no se justifican.

**Especificación formal y desarrollo formal:** se extiende el paso previo haciendo también la justificación.

Entonces, ¿qué nivel se debería adoptar? La mayoría de las experiencias en el uso de métodos formales ha sido en el primer nivel, y ciertamente el procedimiento de especificación formal parece ser extremadamente efectivo. Pero el primer nivel aislado aún deja la brecha entre la especificación inicial y el código. Existe el peligro de que se pierdan algunas de las ventajas de la especificación formal al no ser suficientemente abstractos. Hay por lo tanto muy buenos argumentos para adoptar el segundo nivel. No es necesario adoptar el desarrollo riguroso de manera uniforme.

Muchos de los beneficios del primer nivel residen en descubrir errores –errores, omisiones, contradicciones o ambigüedades en los requerimientos. Muchos de los beneficios del segundo nivel residen en evitar errores: habiendo logrado una buena especificación inicial, se mantiene la conformidad con ella.

El tercer nivel, el de realmente hacer las justificaciones, continúa la tendencia de incrementar sustancialmente el esfuerzo, pero a cambio de un menor retorno en mejora de la calidad.

Convertirse en un buen especificador requiere de cierta experiencia. Convertirse en un buen diseñador riguroso requiere un poquito más. Y hacer justificaciones requiere todavía más.

### Aplicación selectiva de la formalidad

Se puede elegir aplicar métodos formales sólo a algunas partes de un sistema y no a otras. Hay dos formas en las que se puede hacer esto: seleccionar propiedades y seleccionar componentes.

#### *Seleccionar propiedades*

Se puede elegir especificar sólo unas pocas propiedades críticas, como las propiedades de seguridad. En general se logran dos beneficios con esto:

- Comprensión más profunda de la propiedad misma a través de su captura mediante un lenguaje formal.
- Comprensión de cómo los componentes del sistema necesitan interactuar para mantener la propiedad.

#### *Seleccionar componentes*

Se puede elegir especificar sólo ciertos componentes del sistema. Los componentes para los cuales es probable que los métodos formales sean menos útiles incluyen:

- Componentes estándar como bases de datos o sistemas operativos con los cuales tiene que interactuar el sistema.
- Componentes genéricos que el sistema tiene que instanciar. Dentro de esta categoría también se encuentran las interfaces de usuario.
- Componentes cuyo comportamiento no se estima como muy crítico.
- Componentes existentes que no fueron especificados formalmente y que se están adaptando.

Por lo tanto los componentes que se deberían especificar formalmente son aquellos cuya correctitud es crítica y que se están creando desde cero o adaptando.

### 4.2.5. Sistemas formales

Se ha descrito a RAISE como un *sistema formal*. Un sistema formal tiene cuatro componentes esenciales:

- una notación con una sintaxis definida
- un conjunto de reglas para las fórmulas bien formadas
- una semántica
- una lógica

El sistema formal que se describe aquí es RAISE, basado en su lenguaje de especificación, RSL. Un lenguaje de especificación como RSL apunta a hacer posible el razonamiento, y por lo tanto incluye prestaciones (como tipos abstractos y axiomas) que hacen más tratable al razonamiento, y evita características (como punteros) que lo hacen más difícil.

RSL es, sin embargo, un lenguaje de amplio espectro, ya que se intenta que se use no sólo para la especificación inicial sino también para su desarrollo a lenguajes de programación particulares. De allí que incluye algunas prestaciones de bajo nivel como variables con asignaciones y ciclos. Se verá que, si sólo se usan las características aplicativas secuenciales de RSL, el razonamiento es más fácil que cuando se incluyen características imperativas y concurrentes. Por lo tanto, el método promueve (pero no fuerza) que las especificaciones iniciales sean secuenciales y aplicativas para que se pueda razonar inicialmente con cierta facilidad.

### 4.2.6. Relación de implementación de RAISE

Cualquier sistema formal que apunte a proveer un medio para el desarrollo como así también para la especificación debe proveer una noción de implementación. Es decir, si el módulo $A_0$ se desarrolla al módulo $A_1$, debemos saber si $A_1$ es un desarrollo "correcto". Decimos que $A_1$ es correcto si implementa $A_0$, es decir $A_0$ y $A_1$ están en la relación de implementación. En realidad hay unas cuantas variaciones de la noción de implementación; la de RAISE se elige para cumplir dos requerimientos particulares que surgen de los requerimientos del método que se estuvieron examinando. Si $A_1$ implementa $A_0$, se quiere que se cumpla lo siguiente:

**Preservación de propiedades:** todas las propiedades que se pueden probar acerca de $A_0$ también se pueden probar para $A_1$ (pero no viceversa, en general).

**Sustitutividad:** una instancia de $A_0$ en una especificación puede ser reemplazada por una instancia de $A_1$, y la nueva especificación resultante debería implementar la especificación anterior.

La preservación de propiedades asegura que la implementación es transitiva. Para comprender la implementación en la práctica se debe conocer qué se entiende por las propiedades (lógicas) de una expresión de clase. Intuitivamente, es sólo el conjunto de expresiones lógicas que se pueden deducir de sus definiciones y axiomas. Al conjunto de propiedades lógicas de una especificación se le llama la *teoría* de una especificación.

#### La implementación cumple con sus requerimientos

Habiendo provisto cierta intuición para la relación de implementación, se debería comprobar que cumple con los requerimientos de preservación de propiedades y sustitutividad que se identificaron antes.

**Preservación de propiedades:** esto es inmediato de la definición de implementación.

**Sustitutividad:** aquí hay dos preguntas:

- El resultado de un reemplazo en algún contexto, ¿es siempre bien formado?
- El resultado de un reemplazo en algún contexto, ¿siempre da una implementación?

Para responder a la primera pregunta, se debe notar que debido a que la nueva signatura incluye la anterior, el reemplazo de la anterior con la nueva no puede resultar en ningún nombre que quede indefinido. El único problema posible es el de cualquier nombre adicional definido en la nueva.

Primero, se podría obtener un resultado que está mal formado (esto sólo ocurre con la extensión de expresiones de clase). También existe la posibilidad de "captura" de nombres libres. Por lo tanto, sólo se puede permitir el reemplazo si el resultado está bien formado y si no hay captura de nombres libres.

Para responder a la segunda pregunta, se establecen las propiedades "composicionales" requeridas. Se puede mostrar que la implementación tiene estas propiedades.

#### Extensión conservativa

Una teoría *T2* extiende una teoría *T1* si *T2* agrega algo a las entidades y/o propiedades de *T1*. Esto es en realidad lo mismo que la noción de implementación. Se puede distinguir entre *extensión conservativa* y *extensión no conservativa*.

Una teoría *T2* extiende en forma conservativa una teoría *T1* si cada propiedad de *T2* que se puede expresar usando entidades definidas en *T1* es una propiedad de *T1*. Una extensión que no es conservativa es no conservativa.

¿Es importante la extensión conservativa? Con frecuencia no lo es, pero se puede transformar en un asunto importante cuando se realiza desarrollo separado.

### 4.2.7. Ejemplo: Ascensor

#### Objetivos del ejemplo

En [Gro95] se desarrolla como ejemplo la especificación de un sistema simple pero donde la seguridad es crítica, y que involucra concurrencia.

Hay que ser muy cuidadosos al establecer las propiedades de seguridad y al justificarlas. Por lo tanto, es mejor comenzar con una especificación aplicativa y desarrollarla hacia una concurrente. El ejemplo está diseñado para mostrar:

- Cómo especificar axiomáticamente un sistema aplicativo satisfaciendo propiedades de seguridad.

- Cómo desarrollar tal sistema hacia uno con funciones aplicativas explícitas sobre un estado global.
- Cómo descomponer la especificación global aplicativa en componentes aplicativos.
- Cómo obtener un sistema concurrente descompuesto a partir del aplicativo.

En el Apéndice A se describen detalladamente los pasos necesarios para cumplir con los objetivos mencionados.

## 4.3. Métricas Formales

La primera obligación de cualquier actividad de medición de software es identificar las entidades y atributos que se desean medir. En software, existen tres de esas clases [FP96]:

- **Procesos**, conjuntos de actividades relacionadas al software
- **Productos**, artefactos, entregables o documentos que resultan de una actividad del proceso
- **Recursos**, entidades requeridas por una actividad del proceso

A su vez, cada clase de entidad tiene atributos **internos** y **externos**. Los primeros son los que se pueden medir puramente en términos de la entidad misma, sin considerar su comportamiento. Los atributos externos, en cambio, sólo pueden medirse teniendo en cuanta la relación de la entidad con su ambiente, es decir que lo que importa aquí es el comportamiento de la entidad.

La mayoría de las métricas que pueden obtenerse en proyectos de software que utilizan metodologías más tradicionales también son válidas para los métodos formales, aunque por supuesto los valores obtenidos en proyectos similares y para las mismas entidades pueden ser sensiblemente diferentes. Por ejemplo, debido al mayor énfasis que los métodos formales ponen en las primeras etapas (ingeniería de requerimientos, especificación del sistema), cualquier métrica que represente el esfuerzo insumido en cada fase del proceso de desarrollo tendrá valores más elevados en esas primeras etapas, mientras que en una metodología más tradicional, el mayor esfuerzo quizás se insuma en etapas más tardías, como la codificación o la prueba.

Sin embargo, algunos trabajos se han abocado a la investigación de aspectos específicos de los métodos formales, y a la definición de métricas que no se podrían aplicar en otros casos.

### 4.3.1. Aplicación de Métricas a Especificaciones Formales

Está generalmente aceptado que un razonamiento incorrecto en las primeras etapas del desarrollo de software provoca decisiones desacertadas por parte de los desarrolladores, que pueden conducir a la introducción de fallas o anomalías en los sistemas. La mayoría de las decisiones de desarrollo importantes se toman en la etapa temprana de especificación del sistema. Las métricas de software generalmente se recolectan en las etapas de codificación o prueba, cuando ya se sienten las repercusiones de un trabajo mal hecho.

En [VLK98] se presenta un modelo tentativo para predecir aquellas partes de las especificaciones formales que son más propensas a las inferencias erróneas, con el objeto de reducir las fuentes potenciales de errores humanos. Los datos empíricos que alimentaron el modelo se generaron durante una serie de experimentos cognitivos dirigidos a identificar aquellas propiedades lingüísticas de la notación Z que son propensas a admitir errores y prejuicios de razonamiento no lógico en usuarios entrenados.

El modelo toma en cuenta tres factores, que se sugieren como los más influyentes al momento de efectuar un razonamiento:

- El *nivel de experiencia* de los "razonadores" (*Novato* o *Experto*), es decir, el tiempo que llevan los participantes usando la notación Z.
- El *tipo de inferencia* a trazar, relacionado con los constructores lógicos usados en la especificación (*Modus Ponens* y *Modus Tollens*, que permiten llegar a conclusiones válidas; *Negación del Antecedente* y *Afirmación del Consecuente*, que no lo permiten).
- El grado de *contenido significativo* en el material de trabajo, es decir, la medida en la que la especificación es "abstracta", carente de contenido temático o de conceptos realistas (*Temático* o *Abstracto*).

Mediante análisis de regresión sobre los datos del estudio piloto, los autores derivan una función *logit* que es función de los tres factores mencionados anteriormente, y que

da como resultado un valor $z$. Este valor aislado carece de significado, por lo que un último paso lo convierte en probabilidad absoluta, es decir un valor comprendido entre 0 y 1. Ese valor, simbolizado con la letra $p$, provee un medio para que los usuarios de métodos formales puedan predecir la probabilidad de que un razonador con una cierta experiencia realice una inferencia de un tipo dado sobre un tipo dado de declaraciones lógicas expresadas en un cierto grado de material temático. El mencionado valor de $p$ se expresa como

$$p = \frac{e^z}{1 + e^z}$$

donde $z$ es el valor de *logit*, y $e$ es la función exponencial.

A modo de ejemplo, [VLK98] compara los resultados que se obtendrían bajo dos condiciones opuestas. Por un lado un desarrollador *experto*, ante una inferencia de tipo *Modus Tollens* contando con material *temático*, tiene una probabilidad del 95,6 % de llegar a una conclusión lógicamente correcta. En oposición, si una inferencia del mismo tipo pero dada en forma *abstracta* tiene que ser deducida por un usuario *novato*, esa probabilidad se reduce en un 35 %.

El trabajo ilustra entonces que se puede utilizar un enfoque científico para predecir y tratar de disminuir la probabilidad de error en el razonamiento formal.

### 4.3.2. Medición de Especificaciones en Z

Las métricas de software son indicadores cuantitativos útiles para evaluar y predecir atributos de la calidad del software; un atributo que comúnmente se mide es la complejidad del software. Las principales desventajas son: que sólo pueden calcularse luego de realizar un importante esfuerzo de desarrollo para producir el código fuente; que no pueden proveer retroalimentación temprana durante la fase de especificación; y por consiguiente, es costoso introducir cambios al sistema si las métricas así lo indican. A la fecha, hay muy pocos trabajos tendientes a medir la complejidad de un sistema en su fase temprana de especificación [WY04].

En el trabajo mencionado, los autores delinean una serie de métricas para especificaciones escritas en Z. Ésta es una de las notaciones de especificación formal más difundidas, y se basa en el cálculo proposicional y de predicados.

El componente principal en Z es el *esquema*. Para un esquema dado $\sigma_i$ se pueden obtener las siguientes métricas:

- **Métrica 1** cantidad de $\sigma_i$ directamente accedidos por otros esquemas (DAD)
- **Métrica 2** frecuencia de $\sigma_i$ directamente accedidos por otros esquemas (FDAD)
- **Métrica 3** cantidad ponderada de $\sigma_i$ accedidos por otros esquemas (AD)
- **Métrica 4** frecuencia ponderada de $\sigma_i$ accedidos por otros esquemas (FAD)
- **Métrica 5** cantidad de $\sigma_i$ que acceden directamente a otros esquemas (DA)
- **Métrica 6** frecuencia de $\sigma_i$ que acceden directamente a otros esquemas (FDA)
- **Métrica 7** cantidad ponderada de $\sigma_i$ que acceden a otros esquemas (A)
- **Métrica 8** frecuencia ponderada de $\sigma_i$ que acceden a otros esquemas (FA)

Las métricas 1 a 4 describen el reuso de esquemas. Valores mayores llevan a una mayor capacidad de reuso. Cuanto mayores sean los valores, mayor es la probabilidad de lograr una abstracción errónea, y por lo tanto se debe poner más atención a la prueba del esquema.

Las métricas 5 a 8 describen el acoplamiento de esquemas. Cuanto mayores sean los valores, mayor es el número de estructuras y comportamientos que hereda el esquema, y por lo tanto menor es la probabilidad de reuso.

Además, calculan otras 4 métricas:

- $InV$: número de variables de entrada
- $OuV$: número de variables de salida
- $NOAS$: número de declaraciones
- $NOP$: número de predicados

y a partir de este conjunto intentan establecer una relación entre las métricas Z y las métricas CK de Chidamber y Kemerer [CK94] para sistemas orientados a objetos.

A modo de ejemplo, puede citarse el trabajo de Snook y Harrison [SH04], que en base a las métricas mencionadas llevaron a cabo un experimento para probar la hipótesis de que las especificaciones formales no eran más difíciles de leer y comprender que el código correspondiente. Para esto se comparó una especificación en Z con su implementación en Java. El análisis estadístico de los datos recogidos confirma su teoría.

### 4.3.3. Métricas de Cobertura para Verificación Formal

En una verificación formal, se verifica que un sistema es correcto con respecto a una especificación. Incluso cuando se prueba que el sistema es correcto, todavía queda la pregunta de cuán completa es la especificación, y si realmente cubre todos los comportamientos del sistema. El desafío de hacer que el proceso de verificación sea tan exhaustivo como sea posible es aún más crucial en verificación basada en simulación, donde la tarea poco factible de chequear todas las secuencias de entrada se reemplaza por la prueba de un subconjunto finito de ellas.

Es muy importante medir la exhaustividad del conjunto de prueba, y de hecho se ha investigado bastante en la comunidad de verificación basada en simulación sobre esas métricas de cobertura. Pero no hay una única medida que pueda ser absoluta, lo que dio lugar al desarrollo de numerosas métricas de cobertura cuyo uso está determinado por las metodologías de verificación industrial. Por otro lado, la investigación previa sobre cobertura en verificación formal se ha centrado solamente en cobertura basada en estados. En [CKV03], los autores adaptan el trabajo realizado sobre cobertura en verificación basada en simulación al ambiente de la verificación formal con el objeto de obtener nuevas métricas de cobertura.

Los autores dividen su conjunto de métricas de cobertura en aquellas que miden la cobertura sintáctica, y las que miden la cobertura semántica:

**Cobertura sintáctica**

- **Cobertura de código:** las más usadas son la cobertura de sentencias y la cobertura de ramas. Esencialmente, un objeto es cubierto si es visitado durante la ejecución de la secuencia de entrada.
- **Cobertura de circuito:** se refieren al circuito que describe el diseño. Se identifican las partes físicas del diseño que son cubiertas. Esencialmente se mide si una variable cambia su valor durante la ejecución de la secuencia de entrada.
- **Conteo de aciertos:** esta noción reemplaza la indagación binaria de cobertura por medidas cuantitativas (cuántas veces se visitó un objeto). Intuitivamente, cuantas más veces se visita un elemento, mayor es la confianza de que su funcionalidad está probada.

**Cobertura semántica**

- **Cobertura FSM** (Máquina de Estados Finitos): una transición o estado de la FSM abstracta del sistema es cubierto si se lo visita durante la ejecución de la secuencia de entrada.
- **Cobertura de aserciones:** mide qué aserciones son cubiertas por un conjunto dado de secuencias de entrada.

El trabajo además brinda formas algorítmicas para calcular las métricas mencionadas.

### 4.3.4. El Estudio del Caso ATC

A medida que la calidad y la confiabilidad se vuelven factores más importantes en el desarrollo de software debido a un incremento en la competencia del negocio y a distintos requerimientos regulatorios, muchas organizaciones están considerando el uso de métodos formales como parte de su proceso de desarrollo. Mediante el uso de métodos formales, puede demostrarse rigurosamente que una implementación de software satisface su especificación.

El grado de uso de estos métodos puede ser variable. Por un lado, podría desarrollarse formalmente una especificación inicial usando notación matemática y lógica precisa, y luego mostrar mediante pruebas de correctitud que esta especificación es consistente. Luego, el sistema de software podría desarrollarse usando métodos convencionales, o bien se podría realizar un desarrollo formal completo (es decir, la especificación se transforma gradualmente en la implementación formal usando técnicas de refinamiento, y realizando pruebas de correctitud para mostrar que cada paso es un refinamiento válido de la especificación original).

En [SE04] se investigan cuestiones relacionadas con la aplicación de métodos formales al desarrollo de un sistema de información para Control de Tráfico Aéreo (ATC) [Hal96]. Se analiza el proyecto desde el punto de vista de las métricas y la medición, para averiguar cuestiones como si los métodos formales tienen algún efecto sobre la calidad del producto final, y si estos métodos podrían realmente ahorrar costos de desarrollo, en comparación con el desarrollo tradicional.

El sistema desarrollado es el responsable de mostrar información a los controladores del tráfico aéreo. Los controladores interactúan con el sistema para elegir la

información que quieren ver, que incluye vuelos entrantes y salientes, condiciones climáticas, estado del equipamiento en los aeropuertos, y otra información ingresada manualmente al sistema.

La decisión de usar métodos formales fue consecuencia directa de los requerimientos no funcionales del sistema, que especificaban que la información debería ser exhibida dentro de los 1-2 segundos de recibida, la disponibilidad debería ser del 99.97 %, y no debería haber ningún punto individual de falla.

Se usaron métodos formales en la especificación, diseño y verificación. Debido al gran tamaño y complejidad del sistema se usaron varios métodos formales diferentes. También se usaron métodos convencionales. En particular, no se usaron métodos formales durante la fase de implementación.

Luego de haber recogido suficiente información sobre errores y fallas, como así también sobre esfuerzo insumido en las distintas fases del desarrollo, se analizó esa información y se comparó con métricas originadas en proyectos similares. Las conclusiones principales son:

- Mejor relación costo beneficio que los proyectos similares. El proyecto produjo una media de 13 líneas de código entregado por día.
- Menor cantidad de fallas del producto entregado, en comparación con productos similares. Sólo se reportaron 0.75 fallas por KLOC durante los primeros 20 meses de uso.

Sin embargo, estos datos no deberían llevar a la conclusión de que los métodos formales por sí mismos mejoran sustancialmente la calidad de un sistema de software. La diferencia puede deberse además a otros factores, como el proceso de software, el personal, la organización, o el tipo de proyecto. Lo más probable es que la combinación de los métodos formales con los informales, junto con un análisis meticuloso de los requerimientos, resulte en un código de mejor calidad.

# Capítulo 5

# Conclusiones

En este trabajo se exploraron tres metodologías diferentes de desarrollo de software. A pesar de que en esencia todas persiguen el mismo objetivo, el de conducir un proyecto de desarrollo hacia una culminación exitosa, logrando un producto de software correcto y que cumpla con las expectativas del usuario, puede afirmarse que las tres tienen características altamente distintivas, que determinan el tipo de proyectos en los que cada una resulta más conveniente.

## 5.1. Modelo de Desarrollo en Espiral

### Características Principales

El *Modelo de Desarrollo en Espiral* es, como su nombre lo indica, un enfoque cíclico para el desarrollo de productos de software. Las principales características distintivas de este modelo son:

- Tener un enfoque *cíclico* para el crecimiento incremental del grado de definición e implementación de un sistema.
- Ser en realidad un generador de modelos de proceso, o un *metamodelo*, ya que en cada iteración puede contener, dentro de la etapa de construcción, otra metodología más tradicional, como un modelo en Cascada o uno conducido por Prototipos Evolutivos.
- Estar conducido por el *análisis de riesgos*, por lo que intenta identificarlos y minimizarlos en las etapas tempranas de cada ciclo.

Cada ciclo, en la propuesta original, consta de cuatro fases:

- *Identificación* de objetivos, alternativas y restricciones del ciclo.
- *Evaluación* de alternativas, análisis de riesgos, y posible resolución de los mismos.
- *Ingeniería* del producto intermedio (desarrollo y verificación, utilizando un modelo en cascada, o algún otro modelo).
- *Planificación* de la próxima iteración.

El proceso de desarrollo puede verse como un diagrama polar: la dimensión angular corresponde al *progreso* del proyecto en el tiempo, mientras que la dimensión radial representa el *costo* acumulado del proyecto.

El modelo indica lo que debe hacerse a continuación y por cuánto tiempo. Esa determinación varía de acuerdo al proyecto, e incluso de un ciclo de la espiral al siguiente. Cada elección genera un modelo de proceso diferente, y esa elección debe hacerse con la participación activa de todos los interesados en el éxito del proyecto, aportando su visión particular de los riesgos que se deberán afrontar.

La gestión de los riesgos se lleva a cabo mediante un *Plan de Gestión de Riesgos*, que consiste en seguir una serie de pasos:

- *Enumerar los riesgos* del próximo ciclo.
- *Establecer sus prioridades* de acuerdo al grado de importancia, es decir el impacto ponderado por su probabilidad de ocurrencia.
- *Determinar una estrategia de mitigación* para cada uno de los riesgos.

El Modelo en Espiral puede ser adoptado para toda la vida del proyecto. A diferencia de otros ciclos de vida clásicos, este modelo no culmina cuando el software desarrollado se entrega al cliente, sino que permanece en funcionamiento hasta el producto queda obsoleto.

El Modelo en Espiral no es universalmente comprendido. Algunas simplificaciones introducidas en el modelo original han provocado ciertas interpretaciones erróneas, como que la espiral es sólo una secuencia de incrementos en cascada, que todo en el proyecto sigue una única secuencia en espiral, que todos los elementos del diagrama

deben visitarse en el orden indicado, y que no puede haber retrocesos para revisar decisiones previas. Además, suele confundirse al Modelo en Espiral con otros procesos similares pero diferentes en su esencia.

## Ventajas

- Su rango de opciones adopta las características buenas de los modelos existentes de proceso de software, mientras que su enfoque conducido por riesgos evita muchas de sus dificultades.
- Pone atención tempranamente en las opciones que involucran el reuso de software existente.
- Incluye una preparación para la evolución y el crecimiento del ciclo de vida, y para los cambios en el producto de software.
- Provee un mecanismo para incorporar objetivos de calidad del software dentro del desarrollo del producto de software.
- Se concentra en la eliminación temprana de errores y alternativas poco atractivas.
- Por cada una de las fuentes de gastos en recursos y actividad del proyecto, responde a la pregunta clave, "¿cuánto es suficiente?".
- Provee un encuadre viable para el desarrollo integrado de hardware-software.

## Dificultades

- No se corresponde bien con el desarrollo de software por contrato.
- Confía demasiado en la capacidad de expertos en la evaluación de riesgos.
- Los pasos del proceso del Modelo en Espiral necesitan elaboración adicional para asegurar que todos los participantes del desarrollo del software están operando en un contexto consistente.

Una dificultad adicional para aplicar el Modelo en Espiral es su carencia de una guía explícita para determinar, al principio de cada ciclo, los objetivos, restricciones

y alternativas del potencial sistema. Una extensión posterior del Modelo en Espiral, denominada *NGPM* (*Next Generation Process Model*) incluyó como primeras tareas de cada ciclo la identificación de los interesados principales y de sus condiciones de ganancia, para lograr un conjunto de objetivos, restricciones y alternativas que sean satisfactorias para todos.

La extensión NGPM se basa en la *Teoría W* (*win-win*), cuya principal consigna es "*hacer que todos los involucrados ganen*". La intención es generar situaciones donde se privilegien las expectativas e intereses de todas las partes interesadas, evitando situaciones donde alguna de las partes pueda perder.

## Invariantes y Variantes

Hay seis características que siempre están presentes en los ciclos del modelo en espiral, y que se denominan Invariantes:

1. Determinación de artefactos concurrente, más que secuencial.
2. Consideración en cada uno de los ciclos de la espiral de los principales elementos de la misma.
    - Objetivos y restricciones de los interesados más importantes.
    - Alternativas de producto y de proceso.
    - Identificación y resolución de riesgos.
    - Revisión por parte de los interesados.
    - Compromiso para proseguir.
3. Uso de las consideraciones del riesgo para determinar el nivel de esfuerzo que se le debe dedicar a cada actividad dentro del cada ciclo de la espiral.
4. Uso de las consideraciones del riesgo para determinar el grado de detalle de cada artefacto producido en cada ciclo de la espiral.
5. Manejar las metas del ciclo de vida con tres hitos:
    - Objetivos del Ciclo de Vida (OCV).
    - Arquitectura del Ciclo de Vida (ACV).

- Capacidad Operativa Inicial (COI).

6. Énfasis en las actividades y artefactos para el sistema y el ciclo de vida más que para el software y el desarrollo inicial.

Dentro de cada invariante, sin embargo, existe un número de opciones, o variantes, entre las que se puede elegir la más adecuada en función del proyecto actual. La presencia de estos invariantes hace que el Modelo en Espiral se diferencie sustancialmente de otros modelos que por lo demás pueden parecer similares.

Uno de los mencionados invariantes es la inclusión de Hitos de Anclaje, que sirven como puntos de compromiso y de control del progreso del proyecto. Estos hitos son tres, y se enumeran a continuación:

- *Objetivos del Ciclo de Vida* (OCV): es el compromiso de los interesados para respaldar la arquitectura.
- *Arquitectura del Ciclo de Vida* (ACV): es el compromiso de los interesados para mantener el ciclo de vida completo
- *Capacidad Operativa Inicial* (COI): es el compromiso de los interesados para sostener las operaciones.

## Métricas

Las métricas específicas de proyectos que siguen un Modelo en Espiral tienen relación principalmente con la valoración de los riesgos. Como en general esa valoración depende en gran medida de la habilidad de los desarrolladores de software para identificar y manejar las fuentes de los riesgos del proyecto, es importante contar con un modelo de valoración menos subjetivo.

En ese sentido, se ha estudiado un modelo [Nog00b] que se basa en métricas que pueden recolectarse en forma automática, y prácticamente desde el principio del desarrollo. Los factores que se tienen en cuenta son la estabilidad y complejidad de los requerimientos del usuario, la capacidad de adaptación y la eficiencia del equipo, la respuesta y aceptación del usuario, entre otros.

### Aplicación

La naturaleza conducida por riesgos del Modelo en Espiral es más adaptable a todo el espectro de situaciones de los proyectos de software, en comparación con modelos más tradicionales. Este modelo de desarrollo es especialmente aplicable en proyectos complejos y ambiciosos, de grandes dimensiones, de larga duración, con muchas alternativas diferentes, y donde el riesgo de elaborar una solución en base a decisiones equivocadas pueda tener consecuencias desastrosas.

Se han reportado casos exitosos de aplicación del Modelo en Espiral en ámbitos muy diversos, y en [Boe00a] puede verse un resumen de ellos. Allí se mencionan proyectos de software para comercio electrónico, grandes aplicaciones en telecomunicaciones, y proyectos aeroespaciales, entre otros.

## 5.2. Metodologías Orientadas a Objetos

### Características Principales

El *Paradigma Orientado a Objetos* tiene muchas características distintivas, que en sus comienzos marcaron un punto de inflexión en la forma de desarrollar productos de software, y que luego propiciaron un auge muy importante. Algunos de los muchos conceptos que comenzaron a tomar relevancia son:

- *Clase:* es el bloque de construcción más importante de cualquier sistema orientado a objetos; es una abstracción de los datos y procedimientos de una entidad.
- *Objeto:* es una manifestación concreta de una clase.
- *Atributo:* es una propiedad de una clase.
- *Operación:* es un servicio que puede ser requerido de cualquier objeto de la clase.
- *Mensaje:* es el medio a través del cual interactúan los objetos.
- *Encapsulamiento:* los datos y operaciones de una clase están ocultos al mundo exterior.

- *Herencia:* es una relación entre una cosa general (el padre) y un tipo más específico de esa cosa (el hijo).

- *Polimorfismo:* es la propiedad que tienen los métodos de mantener una respuesta unificada, con la misma semántica pero con distinta implementación, a través de la jerarquía de clases.

Los sistemas orientados a objetos tienden a evolucionar con el tiempo, por lo que el mejor paradigma para la Ingeniería de Software orientada a objetos consiste en un modelo de proceso *evolutivo*, acoplado con una estrategia de reutilización de componentes. Este proceso evolutivo consta de varias fases:

- *Comunicación con el usuario*, para obtener una idea clara de sus requerimientos y definir los casos de uso relevantes.

- *Análisis de los requerimientos*, para obtener una definición refinada y estructurada de los mismos.

- *Diseño del sistema*, donde se identifica la arquitectura del sistema y se particularizan los casos de uso.

- *Diseño detallado*, donde se agregan detalles de implementación, adecuando el análisis a las características específicas del ambiente de implementación.

- *Implementación y Pruebas*, donde se desarrolla el código y se comprueba que el sistema funcione correctamente.

Se vio que existen diferentes métodos para el *Análisis y Diseño Orientado a Objetos*, cada uno con una notación específica y con características distintivas. Se mencionaron y analizaron brevemente:

- Desarrollo Orientado a Objetos (OOD), de Booch.

- Diseño Orientado a Objetos Jerárquico (HOOD).

- Técnica de Modelado de Objetos (OMT), de Rumbaugh.

- Diseño Conducido por Responsabilidades (RDD) / Clase-Responsabilidad-Colaboración (CRC), de Wirfs-Brock.

- Análisis Orientado a Objetos (OOA), de Coad y Yourdon.
- Análisis Estructurado Orientado a Objetos (OOSA) / Lenguaje de Diseño Orientado a Objetos (OODL), de Shlaer/Mellor.

Además de éstos, se estudió en detalle el *Proceso Unificado de Desarrollo de Software*, de Jacobson, Booch y Rumbaugh, y el *Lenguaje de Modelado Unificado* (UML).

La *Programación Orientada a Objetos* está caracterizada por un gran número de lenguajes orientados a objetos o basados en objetos. Los más antiguos tienen su origen cuando la Ingeniería de Software aún no era una disciplina reconocida, es decir que la programación orientada a objetos nació antes que cualquier metodología de desarrollo de software orientada a objetos. También se mencionaron y describieron unos pocos de esos lenguajes:

- Simula
- Smalltalk
- Modula-3
- Self
- Eiffel
- Sather
- C++
- Java

Las *Pruebas Orientadas a Objetos* tienen el mismo objetivo fundamental que en cualquier otra metodología, que es el de detectar el mayor número de errores con un esfuerzo y un tiempo razonable. Sin embargo, la naturaleza de los sistemas orientados a objetos cambia la estrategia y las tácticas de las pruebas.

Para probar adecuadamente los sistemas OO, deben hacerse tres cosas:

- La definición de las pruebas debe ampliarse para incluir técnicas de detección de errores aplicables a los modelos de DOO y AOO.

- La estrategia para las pruebas de unidad e integración deben cambiar significativamente.
- El diseño de casos de prueba debe tener en cuenta las características propias del software orientado a objetos.

## Proceso Unificado

Nacido de la necesidad de unificar diferentes técnicas de modelado de objetos, el Proceso Unificado es, junto con su lenguaje de especificación UML, la metodología de desarrollo orientado a objetos más ampliamente difundida y más universalmente adoptada. El Proceso Unificado es un conjunto de etapas parcialmente ordenadas con las que se pretende transformar en forma eficiente y predecible los requerimientos de un usuario en un producto de software que se ajuste a sus necesidades.

Sus tres características principales son:

- Ser dirigido por casos de uso.
- Ser centrado en la arquitectura.
- Ser iterativo e incremental.

Por otro lado, se divide al ciclo de vida en cuatro fases:

- *Inicio*, que apunta a establecer los objetivos del ciclo de vida para el proyecto, y a reducir riesgos.
- *Elaboración*, que pretende delinear la arquitectura, capturar la mayoría de los requerimientos, y reducir más riesgos.
- *Construcción*, que intenta desarrollar el sistema completo y desarrollar su capacidad operativa inicial.
- *Transición*, que trata de asegurar que se tiene un producto listo para ser entregado a los usuarios.

En cada una de estas fases varía la cantidad de esfuerzo dedicado a las distintas actividades típicas de captura de requerimientos, análisis, diseño, implementación y pruebas. Durante las fases de inicio y elaboración, la mayor parte del esfuerzo

está dirigida a la captura de los requerimientos y al análisis y diseño preliminar. Durante la construcción el énfasis se desplaza hacia el diseño, implementación, y prueba detallados. Las primeras fases tienen mucho de gestión del proyecto, y de desarrollo de un ambiente para el proyecto.

## UML

El UML es un lenguaje estándar para la escritura de diseños de software. El UML se puede usar para visualizar, especificar, construir, y documentar los artefactos de un sistema de software. Es un lenguaje muy expresivo, por lo que resulta adecuado para un rango muy amplio de sistemas.

Para comprender el UML es necesario formarse un modelo conceptual del lenguaje, y esto requiere el aprendizaje de tres elementos principales: los *bloques de construcción* básicos del UML (cosas, relaciones, y diagramas), las *reglas* que dictan cómo se pueden reunir esos bloques de construcción (reglas de nombres, de alcance, de visibilidad, de integridad, y de ejecución), y algunos *mecanismos comunes* (especificaciones, adornos, divisiones comunes, y mecanismos de extensibilidad) que se aplican a lo largo de todo el UML.

## Evaluación y Aplicación

Las Metodologías Orientadas a Objetos, en general, tienen tres ventajas principales:

- Las tecnologías de objetos llevan a reutilizar y la reutilización de componentes de software conduce a un desarrollo de software más rápido y a programas de mejor calidad.
- El software orientado a objetos es más fácil de mantener debido a que su estructura es inherentemente descompuesta. Esto provoca menores efectos colaterales cuando se deben hacer cambios.
- Los sistemas orientados a objetos son más fáciles de adaptar y escalar.

El Proceso Unificado ha demostrado ser una metodología eficiente, pero para algunos es demasiado "burocrática", es decir, requiere una documentación exhaustiva en muchos casos difícil de llevar.

En el caso de grandes proyectos, sin duda el Proceso Unificado junto con UML son una excelente opción, siempre y cuando los involucrados en el proyecto comprendan bien de qué se trata el asunto. El Proceso Unificado no es una receta para hacer software, es una metodología muy seria acerca del proceso de creación del software, y como la mayoría de las cosas serías, puede llegar a ser difícil de aprender. En el otro extremo, para proyectos pequeños o empresas que no cuentan con un gran departamento de desarrollo, el Proceso Unificado es una alternativa costosa tanto en dinero como en tiempo.

Una de las características que hacen del Proceso Unificado una metodología muy exitosa es sin duda su enfoque iterativo, esto es, la metodología parte del supuesto de que se trabajará en iteraciones cortas en tiempo y con metas muy claras. Cada iteración tiene entregables claros y en la medida de lo posible, el sistema debe ser funcional desde las primeras iteraciones de desarrollo. Si una funcionalidad no puede terminarse en una iteración, se terminará la iteración sin ella, es más importante terminar la iteración con un sistema funcional y sin errores o detalles incompletos que un sistema que no puede ser evaluado.

En lo que se refiere específicamente al UML, y a pesar de su universalidad, también tiene puntos criticables, ya desde su propia definición, es decir el metamodelo de UML, que tiene fallas de varios tipos. Además, se critica la informalidad de UML, su dificultad para especificar comportamiento, problemas de comunicaciones y temporización en el ámbito de los sistemas de tiempo real, su complejidad, y que dificulta la reversibilidad.

## Métricas

Los elementos a medir en proyectos orientados a objetos, además de los que se miden en los proyectos de software en general, son los relacionados con las características propias tanto del proyecto, como la determinación de casos de uso o de clases clave, como del producto, como la localización y el ocultamiento de la información, el encapsulamiento, la herencia y la abstracción de datos.

Existen muchos trabajos dedicados a describir métricas de sistemas y de productos orientados a objetos, algunos de los cuales gozan de larga fama:

- El Conjunto de Métricas CK, de Chidamber y Kemerer [CK94].

- El trabajo sobre Métricas Orientadas a Objetos de Lorenz y Kidd [LK94].
- Las Métricas Orientadas a Objetos propuestas por Abreu y Melo [BeAM96].

## 5.3. Métodos Formales

### Características Principales

Por último, los métodos formales de desarrollo presentan una alternativa diferente al proponer un desarrollo incremental basado fuertemente en la lógica formal. El uso de los métodos formales apunta al aumento de la confiabilidad del software. Su principal idea es que debería ser posible razonar sobre las propiedades del software o de los sistemas que incluyen software.

Los métodos formales se interesan principalmente en la *especificación* del software, y en asuntos directamente relacionados con ella. Es decir, en desarrollar una declaración precisa de *qué* debe hacer el software, y no de *cómo* se va a hacer. Se intenta proveer una descripción completamente rigurosa de los *resultados* que debe devolver un elemento de software. Como que estas descripciones de resultados son anteriores a la construcción y ejecución del software, lograr precisión y rigor requiere el uso de formalismos matemáticos/lógicos. Estas especificaciones independientes de la plataforma sirven como un contrato técnico inicial entre el programador y el cliente, y luego guían la creación, verificación, y documentación del software.

Como una especificación provee un contrato técnico, es natural basar tanto la construcción como la verificación de un programa en su especificación. Como resultado, los métodos formales incluyen:

1. elementos conceptuales para el desarrollo de especificaciones precisas que puedan servir de guía a la actividad de programación,
2. los medios para utilizar una especificación formal para una verificación rigurosa del programa cuando éste está completo, y
3. la integración de estas ideas en un "sistema de especificación" que puede apoyarse en herramientas computarizadas que asistan a todo el proyecto.

Cuando se usa un método formal:

- La especificación es formal y ejecutable, y constituye el primer prototipo del sistema. La especificación puede validarse mediante prototipado.
- Luego, a través de transformaciones formales, la especificación se convierte en la implementación del sistema, y en el último paso de transformación se obtiene una implementación en un lenguaje de programación determinado.
- El mantenimiento se realiza sobre la especificación (no sobre el código fuente), la documentación es generada automáticamente y el mantenimiento es realizado por repetición del proceso (no mediante parches sobre la implementación).

Este tipo de metodologías tiene mucha menos difusión que las orientados a objetos. Existe un conjunto de creencias generalizadas (y mayormente erróneas) que obstaculizan una aplicación más generalizada de los métodos formales. Del estudio de esas creencias surgen las siguientes conclusiones:

- Los métodos formales funcionan y son importantes para mejorar la correctitud del software, pero no son infalibles. Hay cosas que no se pueden probar, o se pueden cometer errores en las pruebas.
- Los métodos formales son beneficiosos para todo tipo de proyectos de software, incluso aquellos donde la seguridad no sea crítica.
- Los métodos formales no son fáciles de aplicar, pero tampoco son extremadamente difíciles.
- Los métodos formales no hacen más costoso al proyecto. Por el contrario, pueden reducir su costo en ciertas ocasiones.
- Una especificación formal es incomprensible para un usuario común, pero existen medios para hacer que la especificación se acerque al usuario.
- Los métodos formales no se limitan a ambientes académicos y de investigación, sino que también se usan en proyectos reales, industriales, de grandes dimensiones.
- Los métodos formales no retrasan el proceso de desarrollo. Sólo modifican la duración de cada etapa. Se le da más tiempo a las primeras etapas, pero las últimas, como la de codificación, son más cortas.

- Existen herramientas de soporte que permiten incrementar la productividad y precisión en el desarrollo formal.
- Los métodos formales no reemplazan a los métodos de diseño tradicionales de la ingeniería de software, sino que deberían integrarse para lograr un método de desarrollo "verdadero".
- Los métodos formales también pueden aplicarse al desarrollo de hardware.
- En ocasiones, los métodos formales pueden ser excesivos, pero en otros casos son muy deseables, sobre todo si la seguridad es crítica o si se quieren evitar las consecuencias desastrosas de una falla.

También se determinaron diez lineamientos básicos que se deben tener en cuenta al momento de aplicar métodos formales:

- Elegir una notación apropiada.
- Formalizar, pero no formalizar de más.
- Estimar costos.
- Tener un gurú en métodos formales a quien llamar.
- No abandonar los métodos de desarrollo tradicionales.
- Documentar lo suficiente.
- No comprometer los estándares de calidad.
- No ser dogmático.
- Probar, probar, y volver a probar.
- Reusar.

## Ventajas

Los métodos formales brindan sus mayores ventajas en cuestiones relacionadas con la garantía de calidad del producto. Las principales son:

- Aumento de la comprensión del sistema modelado.

- Automatización de actividades comunes del desarrollo de software (generación de código, por ejemplo).
- Análisis de las implementaciones (detección de errores, ambigüedades, etc.).
- Análisis/simulación de modelos desde fases tempranas del desarrollo.
- Transformación de modelos y comprobación de coherencia entre modelos.

## Limitaciones

- Los métodos formales están poco difundidos, en parte debido a falta de educación al respecto.
- Los métodos formales se pueden usar para verificar un sistema, pero no para validarlo. La diferencia es que la validación muestra que un producto satisface su misión operativa, mientras que la verificación muestra que cada paso del desarrollo satisface los requerimientos impuestos por los pasos previos. En otras palabras, los métodos formales pueden probar que una implementación satisface una especificación formal, pero no que una especificación formal captura la comprensión informal intuitiva que el usuario tiene de un sistema.
- Los métodos formales pueden verificar que una implementación satisface una especificación cuando corre en una máquina abstracta idealizada, pero no cuando corre en cualquier máquina física.
- Requieren desarrolladores especializados y experimentados en este tipo de procesos para llevarse a cabo.

## Lenguajes y Métodos Formales

Cuando se habla de lenguajes formales, existen algunos enfoques que utilizan el mismo lenguaje para escribir desde las especificaciones de alto nivel hasta los programas ejecutables, pasando por todo el proceso de transformación. Ese único lenguaje se denomina *de amplio espectro*.

Algunos enfoques también aseguran que la especificación inicial debe ser ejecutable, para asegurar que refleje fielmente las intenciones del cliente. Un inconveniente

de esto es que se deben tomar decisiones de diseño que de otro modo podrían dejarse abiertas para considerarlas más adelante.

Entre los lenguajes introducidos brevemente se encuentran:

- VDM (*Vienna Development Method*)
- Z
- Lenguajes basados en Reglas de Reescritura
  - HOPE
  - ML Estándar
  - OBJ2
- Lenguajes de Especificación Algebraica
  - CLEAR
  - Larch
  - ML Extendido

También se mencionan las estrategias de refinamiento de especificaciones que adoptan algunos de los métodos mencionados, en particular VDM, Z, y los enfoques algebraicos.

El método RAISE, que se estudia más en detalle en este trabajo, abarca:

- La formulación de especificaciones abstractas.
- El desarrollo de éstas hacia especificaciones sucesivamente más concretas.
- La justificación de la correctitud del desarrollo.
- La traducción de la especificación final a un lenguaje de programación.

Los cuatro principios básicos que rigen el desarrollo de un sistema de software utilizando el método RAISE son:

- *Desarrollo separado:* implica la posibilidad de descomponer la descripción de un sistema en componentes, y luego componer el sistema desde los componentes desarrollados.

- *Desarrollo por etapas:* asume que cada componente se desarrolla en varios pasos de diseño, donde cada uno aborda diferentes decisiones de diseño.

- *Inventar y verificar:* es un estilo que fuerza al desarrollador a inventar un diseño nuevo, para luego verificar su correctitud. Esto lo diferencia de las técnicas basadas en transformaciones, donde el desarrollador parte de una expresión y le aplica una regla de transformación que crea una expresión nueva pero equivalente.

- *Rigor:* No es necesario probar formalmente todas las propiedades de un sistema. Un argumento riguroso es aquel que contiene alguna etapa informal.

La aplicación de RAISE a la ingeniería de software afecta principalmente a las siguientes actividades del proceso de desarrollo:

- Validación y Verificación
- Análisis de Requerimientos
- Mantenimiento de la Correctitud
- Descubrimiento de Errores
- Producción de Documentación

El método RAISE permite un *uso selectivo* de la formalidad. Esto puede hacerse de dos maneras:

- Eligiendo el *grado de formalidad* que se elige tener.
- Eligiendo a qué *componentes o propiedades* se le aplica esa formalidad.

El Lenguaje de Especificación de RAISE, o RSL, es un lenguaje de amplio espectro, es decir que puede usarse no sólo para la especificación inicial sino también para su desarrollo a lenguajes de programación particulares. Por ello incluye algunas prestaciones de bajo nivel, que deberían comenzar a usarse en las especificaciones finales porque de usarse al principio complicarían su comprensión.

Mediante un ejemplo completamente desarrollado, se muestra la aplicación del método RAISE mediante el uso de RSL para la construcción de un sistema de control para un ascensor. El ejemplo muestra:

- Cómo especificar axiomáticamente un sistema aplicativo satisfaciendo propiedades de seguridad.
- Cómo desarrollar tal sistema hacia uno con funciones aplicativas explícitas sobre un estado global.
- Cómo descomponer la especificación global aplicativa en componentes aplicativos.
- Cómo obtener un sistema concurrente descompuesto a partir del aplicativo.

## Métricas

Las métricas orientadas a métodos formales ponen énfasis en detectar fuentes de errores o ambigüedades lo más temprano posible. Se miden las especificaciones en sí, y también factores relacionados con la verificación formal de las mismas.

La mayoría de las métricas que pueden obtenerse en proyectos de software que utilizan metodologías más tradicionales también son válidas para los métodos formales, aunque por supuesto los valores obtenidos en proyectos similares y para las mismas entidades pueden ser sensiblemente diferentes. Por ejemplo, cualquier métrica que represente el esfuerzo insumido en cada fase del proceso de desarrollo tendrá valores más elevados en esas primeras etapas (ingeniería de requerimientos, especificación del sistema) debido al mayor énfasis que los métodos formales ponen en las mismas, en comparación con metodologías más tradicionales, donde el mayor esfuerzo quizás se insuma en etapas más tardías, como la codificación o la prueba.

En lo que se refiere a métricas específicas, se han mencionado:

- Un trabajo que presenta un modelo tentativo para predecir aquellas partes de las especificaciones formales que son más propensas a las inferencias erróneas, con el objeto de reducir las fuentes potenciales de errores humanos.
- Un trabajo donde se propone una serie de métricas que describen el reuso y el acoplamiento en especificaciones escritas en lenguaje Z.
- Un trabajo que describe un conjunto de métricas de cobertura, es decir que miden el grado de completitud al momento de efectuar la verificación formal de

un sistema. Un trabajo donde se describen los resultados de aplicar métodos formales a un sistema de Control de Tráfico Aéreo.

## Aplicación

El rigor de los métodos formales hace que sea especialmente aplicable en sistemas donde la seguridad es un factor crítico, es decir donde una falla en el software puede implicar peligro para vidas humanas o pérdidas económicas cuantiosas, como sistemas de transporte (aviones, trenes, subtes), sistemas de centrales o de transmisión eléctrica, redes de telecomunicaciones, etc. La mayor inversión de recursos en la especificación precisa del sistema se ve recompensada generalmente al final, con un producto eficiente y correcto.

Es frecuente que se introduzcan métodos formales de desarrollo luego de la ocurrencia de un problema grave. Es decir, cobran relevancia cuando se hacen evidentes sus ventajas, o cuando se materializan los problemas de los métodos alternativos.

# Apéndice A

# Ejemplo: Ascensor

En la Sección 4.2.7 se introdujeron los objetivos de este ejemplo. A continuación se describen en detalle los pasos necesarios para cumplir con dichos objetivos.

## A.1. Requerimientos

Se requiere que un ascensor atienda un número de pisos. Cada piso tiene puertas que sólo deben estar abiertas cuando el ascensor está estacionado en ese piso. Cada piso excepto el superior tiene un botón que le ordena al ascensor parar ahí y luego ir hacia arriba; cada piso excepto el inferior tiene un botón que le ordena al ascensor parar ahí y luego ir hacia abajo. El ascensor también tiene un botón por cada piso para ordenarle ir hasta ese piso.

### Simplificación de suposiciones

- No se distingue entre puertas del ascensor y puertas de los pisos. Esto refleja que, o bien el ascensor no tiene puertas, o la puerta del ascensor y la de un piso son obligadas por hardware a que sólo se abran y se cierren juntas (cuando el ascensor está parado en ese piso).
- Se considera que las puertas en cada piso sólo pueden estar en uno de dos estados: "abierta" (cuando el ascensor debe estar en ese piso y estacionado) y "cerrada" (cuando el ascensor puede estar en cualquier otro sitio y/o moviéndose).

- No se considera el tiempo que le toma al ascensor moverse o a las puertas abrirse o cerrarse. Sin embargo, en un nivel detallado, se tendrá eventos de "hacer" y "reconocer" para tales acciones y se asumirá que el hardware nos dice mediante reconocimientos cuando las acciones están completas.
- No se consideran las luces en los botones ni las señales auditivas de que el ascensor está parado en un piso. Se asume que esto se puede hacer puramente por hardware.
- Se hacen algunas suposiciones sobre la forma en que se controla el motor del ascensor — éstas se describirán más adelante.
- No se considera cómo se tratan las fallas de hardware, o cómo reiniciar el sistema luego de tales fallas.
- Se asume que los pisos están numerados en forma consecutiva.

## A.2. Formulación inicial

Un ascensor es un ejemplo de sistema *asincrónico*, ya que los botones pueden ser pulsados en cualquier momento. En otras palabras, hay estímulos externos que pueden ocurrir en cualquier momento, o pueden no ocurrir nunca. Es importante que tales sistemas sean "débilmente acoplados". No debe llegarse a una situación donde el ascensor esté esperando que se pulse un botón mientras el usuario está tratando de pulsar otro.

Con este estilo de desarrollo, este problema se maneja en forma bastante natural. Habrá un módulo *botón* con funciones que le permitan al usuario presionarlo y al ascensor comprobar si ha sido pulsado y liberarlo. Cada botón se modela como un proceso separado, por lo que no hay sincronización entre los usuarios que presionan botones y el ascensor que los inspecciona y libera.

### Componentes del sistema

Se comienza considerando los objetos del sistema y si tendrán estado dinámico:

- El ascensor en sí presumiblemente cambiará su posición, dirección y velocidad mediante comandos a su motor.

- Las puertas estarán abiertas o cerradas.
- Los botones estarán pulsados (e iluminados) o liberados (y apagados).
- Un piso podría ser dinámicamente "visitado" por el ascensor o no, pero esto duplicaría la posición del ascensor. Por lo tanto los pisos sólo parecen tener atributos estáticos, como su número, si están encima o debajo de otros pisos, y si son el piso superior o el inferior.

Ciertamente parece ser que el motor del ascensor, las puertas y los botones tendrán estados dinámicos y por lo tanto se modelarán como objetos RSL.

Se puede construir un diagrama entidad relación (Figura A.1) que ilustre las entidades físicas del sistema.

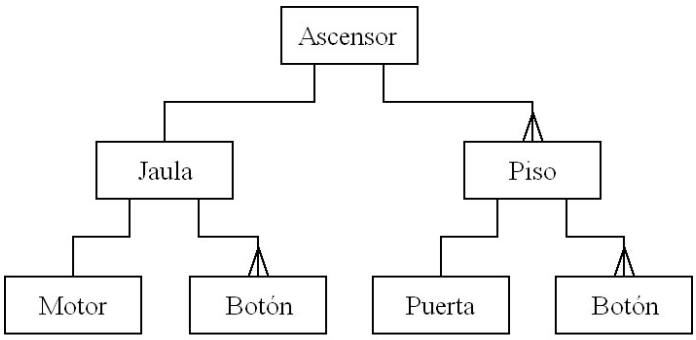


Figura A.1: Relaciones entre las entidades físicas

No se incluye una puerta para la jaula del ascensor debido a la suposición de que este no tiene puerta, o bien es controlada por la puerta del piso.

Para la especificación se adoptará una estructura diferente entre módulos, ya que no existe la restricción de la estructura física: se eliminan los niveles intermedios "Jaula" y "Piso", fusionando los conjuntos de botones.

Entonces se pueden dibujar los componentes propuestos en la especificación como en la Figura A.2, donde sólo se muestran las funciones de generación.

Hasta aquí no es claro cuáles son las funciones externas de los objetos componentes. Ciertamente, debe ser posible pulsar cualquier botón. Entonces el ascensor, ¿se comporta independientemente del control externo (siempre que no haya fallas)? ¿O es necesario seguir diciéndole que ejecute la próxima acción? Por ahora se asume lo último, pero se volverá sobre este asunto más adelante.

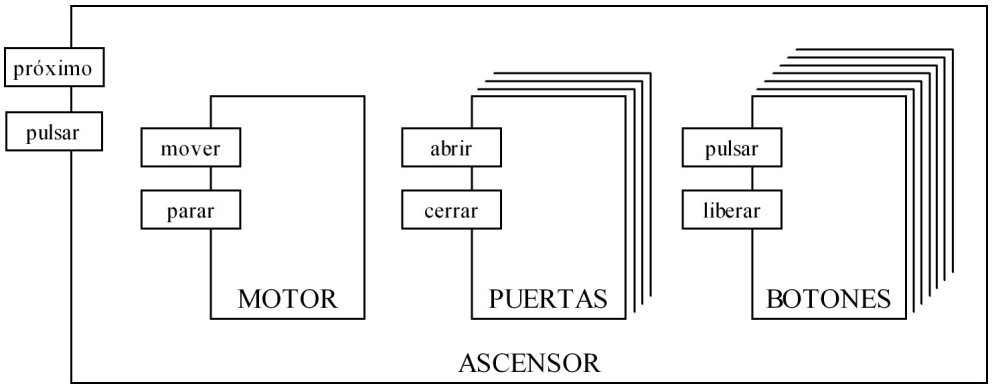


Figura A.2: Componentes de la Especificación

Luego viene la pregunta de qué atributos son necesarios para estos objetos. En este caso está la cuestión de cuán finamente se necesitan modelar las cosas. Las puertas, ¿sólo están abiertas o cerradas, o también tienen estados intermedios de apertura y cierre? ¿Es necesario ir más allá y medir su separación actual, sus velocidades y aceleraciones? Preguntas similares se aplican a los movimientos del ascensor.

Las respuestas a tales preguntas pertenecen a los requerimientos detallados (o deberían aclararse antes de comenzar, si no se establecieron allí). Aquí sólo se distinguirá una puerta que está "cerrada" (que significa cerrada y bloqueada) de una que está "abierta", y se pedirá que una puerta esté abierta en un piso sólo si el ascensor está estacionado en ese piso. (Por lo tanto se ignora, por ejemplo, la necesidad de llamar a un técnico para que destrabe la puerta manualmente si el motor falla.) En la Figura A.3 se muestran las transiciones de estado para cada puerta.

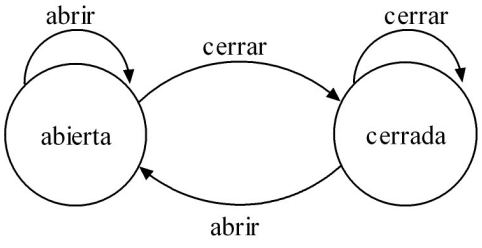


Figura A.3: Transiciones de estado para una puerta

Acerca del ascensor se hacen suposiciones similares. Se asume que puede caracterizarse suficientemente si se dice que puede estar parado en un piso (cuando las puertas deben estar abiertas), o en algún otro estado que se llamará "en movimiento". Cuan-

do está parado estará en un piso; parece conveniente asociar siempre al ascensor con un piso aún cuando se esté moviendo, y éste será el (próximo) piso hacia el cual se está moviendo. Cuando se está moviendo, debe tener una dirección, arriba o abajo. De nuevo parece conveniente asociarle una dirección al ascensor cuando está parado, que es la dirección en la cual se estaba moviendo antes de detenerse. En la Figura A.4 se muestra un diagrama de transición de estados para el ascensor.

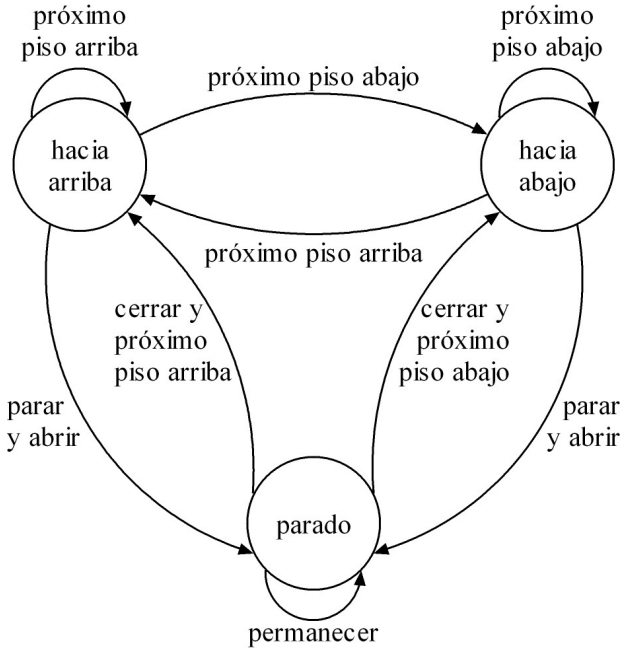


Figura A.4: Transiciones de estado para el ascensor

## Módulo de tipos

La discusión anterior es suficiente para formular el módulo de tipos para el sistema, que se llamará *TIPOS* (como se muestra en el Cuadro A.1) y se instanciará como el objeto global *T*.

---

**scheme** TIPOS =
  **class**
    **value**
      min_piso, max_piso : **Int**,
      es_piso : **Int** → **Bool**

```
    es_piso(f) ≡ f ≥ min_piso ∧ f ≤ max_piso
  axiom [algunos_pisos] max_piso > min_piso
  type
    Piso = {|n : Int • es_piso(n)|},
    Piso_inf = {| f : Piso • f < max_piso |},
    Piso_sup = {| f : Piso • f > min_piso |},
    Estado_puerta = = abierta | cerrada,
    Estado_boton = = iluminado | liberado,
    Direccion = = arriba | abajo,
    Movimiento = = parado | moviendo,
    Requerimiento :: aquí : Bool después : Bool antes : Bool
  value
    prox_piso : Dirección × Piso ~→ Piso
    prox_piso(d, f) ≡
      if d = arriba then f + 1 else f – 1 end
      pre es_prox_piso(d, f),
    es_prox_piso : Direccion × Piso → Bool
    es_prox_piso(d, f) ≡
      if d = arriba then f < max_piso else f > min_piso end,
    invertir : Direccion → Direccion
    invertir(d) ≡ if d = arriba then abajo else arriba end
end
```

Cuadro A.1: Módulo de Tipos para el Sistema

No hay indicación de que los pisos tendrán otros atributos aparte de su número. Se eligió modelar el tipo *Piso* directamente como un subtipo de **Int**. Se asume que los pisos están numerados en forma consecutiva.

El tipo *Requerimiento* se explicará más adelante cuando se discuta cómo se chequean los botones.

## Propiedades de seguridad y existencia

Para un sistema de control como un ascensor, usualmente hay dos tipos de propiedades que interesan. El primer tipo es el de las *propiedades de seguridad*. Una propiedad de seguridad establece que alguna situación nunca debe ocurrir, lo que se puede formular como "el predicado que describe la situación siempre es falso". Las propiedades de seguridad en general no son difíciles de formular en RSL.

El segundo tipo de propiedad es el de las *propiedades de existencia*. Una propiedad de existencia dice que eventualmente algo debe suceder. Desafortunadamente, tales propiedades son en general más difíciles de expresar en RSL, pero se puede proveer un sustituto efectivo, como se verá más adelante.

## Generadores

Como se dijo anteriormente, se está en presencia de un sistema asincrónico, con mensajes que entran en momentos y con frecuencias que están más allá de la influencia del controlador del ascensor, que es la parte que se quiere especificar. Es necesario asegurar que el controlador compruebe si hay nuevos mensajes con la suficiente frecuencia. Por ejemplo, un requerimiento detallado para un ascensor podría ser que si abandona un piso porque existe una orden de ir a algún piso inferior, y alguien presiona el botón de subir en algún piso intermedio antes de que el ascensor lo pase, el ascensor debería detenerse allí. Una forma estándar de diseñar tal sistema de control es con un pequeño ciclo:

**while true do**
  leer mensajes ;
  efectuar la próxima acción adecuada
**end**

donde la "pequeñez" se mide por el monto de "acción" que puede tener lugar durante cada ciclo. En este caso la diferencia entre los pisos actuales antes y después de la próxima acción estará limitada a un máximo de uno, para lograr el requerimiento detallado recién mencionado. Esto significa que se asume que el motor del ascensor es capaz de realizar un viaje directo de varios pisos a pesar de que sólo se dan instrucciones de moverse un piso por vez.

Esta idea estándar de un ciclo de control también sugiere qué deberían ser los

generadores: deberían corresponder a “leer mensajes” y “efectuar la próxima acción”. El primero será un generador que devuelve un resultado; dirá algo acerca del estado de los botones y también cambiará el estado. Puede no ser claro por qué esa función debería generar un nuevo estado, pero esa es la forma de modelarlo. Es como si existe el conjunto de todas las posibles activaciones futuras de botones codificadas en este estado, y esta función lee el próximo grupo y cambia el conjunto para generar el conjunto restante. Dejar sin especificar cuál es el resultado de esta función y cuáles son las características del nuevo estado significa que se comporta tal como un sensor. Esta es la forma en que se deberían modelar aplicativamente todas estas funciones sensoras.

Es necesario considerar cuál debería ser el tipo del resultado de esta función. ¿Un arreglo de valores lógicos para todos los botones? El ascensor necesitaría realizar algún procesamiento sobre esto: si debería parar en el piso hacia el que se está moviendo, por ejemplo, depende de si el botón del ascensor para ese piso está encendido, o si el botón en el piso en la dirección actual está encendido, o si el botón en el piso en la otra dirección está encendido y no hay solicitudes de ir más allá en la dirección actual. Claramente, todo el procesamiento de este tipo puede hacerse en otra parte, y se decide que la función de sensor de botón, llamada *chequear_botones*, devolverá un valor de tipo *Requerimiento*, que es un arreglo de tres valores lógicos que representan si el ascensor es requerido *aquí*, *después* (es decir, en un piso en la misma dirección) o *antes* (o sea, en un piso en la dirección contraria). Se puede asegurar que estos predicados son falsos en los casos de frontera (como *después* en el piso superior). El cálculo de estos valores partiendo de los botones necesitará el piso y la dirección actuales, pero estos probablemente se puedan obtener del tipo de interés *Ascensor*. Por lo tanto, se tiene una signatura para *chequear_botones*:

**value**

chequear_botones : Ascensor → T.Requerimiento × Ascensor

(Recordar que $T$ es la instancia global de *TIPOS*.)

Ahora se necesita una signatura para *próximo*, el otro generador sugerido por el ciclo de control. Esto probablemente necesite los resultados de *chequear_botones*, por lo que se tiene

**value** proximo : T.Requerimiento × Ascensor $\tilde{\to}$ Ascensor

*chequear_botones* puede ser total por lo que no hay necesidad de precondiciones. No

es claro aún qué debería hacer *próximo* si, digamos, *después* es verdadero pero ha alcanzado el piso superior. Así que por ahora se la dejará parcial. Pero se diferirá la definición de su precondición.

## Observadores

Se agregan los observadores:

**value**

movimiento : Ascensor → T.Movimiento,
estado_puerta : Ascensor → T.Piso → T.Estado_puerta,
piso : Ascensor → T.Piso,
direccion : Ascensor → T.Direccion

## Axiomas

El método normal consiste en tratar luego de definir los axiomas para observadores y generadores, pero en este caso se interpondrá un paso adicional. El propósito inicial es capturar las propiedades críticas del ascensor. Estas son:

- una propiedad de seguridad de que las puertas están siempre cerradas si el ascensor no está parado en un piso (para que la gente fuera del ascensor no pueda caer por el agujero), y abiertas si el ascensor está parado en un piso (para que la gente dentro del ascensor pueda salir)
- Una propiedad de existencia de que el ascensor hace algo útil, que eventualmente llegará a un piso y se detendrá allí si se lo requirió.

La propiedad de seguridad puede expresarse mediante una función "invariante" *seguro*, con definición

seguro(s) ≡
(∀ f : T.Piso •
(estado_puerta(s)(f) = T.abierta) =
(movimiento(s) = T.parado ∧ piso(s) = f))

Como ya se dijo, las propiedades de existencia son más difíciles de especificar. Lo que se puede hacer, en lugar de tratar de describir qué significa "eventualmente", es describir alguna relación entre un estado del ascensor y el siguiente. De hecho, esta

será una relación sobre tres estados: el inicial, el estado luego de *chequear_botones*, y el estado luego de *proximo*.

A esos estados se los llama *s*, *s'* y *s"*. Se expresarán las propiedades de que

- Si *s* es seguro, también lo son *s'* y *s"*.
- Si el ascensor está detenido en estado *s"*, este fue requerido *aquí* o en ningún otro lugar, y el piso de *s"* es el mismo que el piso de *s*.
- Si el ascensor no está detenido en estado *s"*, fue requerido o *antes* o *después*, y el piso hacia el que se está moviendo es próximo al piso en estado *s* y un piso válido.
- Si el ascensor ha cambiado de dirección entre los estados *s* y *s"*, *después* debe ser falso.

La primera de estas es la propiedad de que el invariante *seguro* se mantiene. Se podría especificar como un axioma separado, pero es conveniente en este caso incluirlo como parte del axioma de existencia. Si se lo separa, igual habría que usar *seguro(s)* como una precondición del axioma de existencia.

Otro enfoque para establecer los requerimientos de existencia es conformar alguna medida de cuán cerca está un estado de satisfacer la propiedad requerida y luego mostrar que el próximo estado reduce estrictamente esta medida. Esto puede ser muy difícil de hacer cuando, como en este caso, cada parte "leer mensajes" del ciclo cambia lo que se requiere. El argumento de que el ascensor eventualmente alcanzará un piso para el cual hay un botón pulsado es el siguiente:

- Si el ascensor está actualmente detenido en algún otro lugar, *aquí* para ese piso debe ser verdadero. Si *aquí* eventualmente se vuelve falso, o *antes* o *después* debe ser verdadero y el ascensor debe pasar a un estado no detenido (que se espera que sea moviéndose físicamente). Pero nótese que se hace la suposición de que *aquí* debe hacerse falso. Podría ser tentador especificar esto (sobre la suposición de que el ascensor libera los botones relevantes cuando se detiene en un piso) pero no se puede dar garantías de que alguien no vaya a presionar uno de ellos de nuevo inmediatamente. De hecho, nunca se puede predecir cuál será el resultado de *chequear_botones* (o se debería restringir la posibilidad de

que las personas pulsen los botones). Se podría especificar a este nivel que los botones relevantes están liberados, por supuesto, pero

- complicaría la especificación ya que implica definir qué son esos botones, y
- no ayudaría a especificar seguridad o existencia, como se ha visto

por lo que se prefiere dejarlo como un requerimiento que debe ser satisfecho más adelante.

De manera similar, se asume que sólo el ascensor será capaz de liberar un botón, y por lo tanto un botón, una vez presionado, permanecerá encendido. A su vez esto significa que *antes* y *después*, una vez que se hacen verdaderos debido a un botón pulsado, permanecerán verdaderos hasta que el ascensor alcance el piso relevante o invierta su dirección, cuando *antes* toma el valor de *después* y viceversa.

Nunca es posible garantizar absolutamente que incluso un ascensor en funcionamiento va a hacer eventualmente cualquier cosa. Alguien puede mantener las puertas abiertas en algún lugar indefinidamente, por ejemplo. (Sin embargo, se pueden especificar sistemas que detecten tales eventos y pasen a estados especiales cuando sucedan. Entonces las propiedades de existencia incluyen suposiciones sobre los estados "normales" como así también sobre los "seguros".)

- Si se acepta la suposición de que el ascensor no va a permanecer detenido indefinidamente en ningún lugar, entonces se va a mover hacia el "próximo" piso. Allí puede detenerse pero, de nuevo, eventualmente se moverá. Por lo tanto se sabe que el ascensor siempre "hace progresos".

  - Si el movimiento es hacia el piso en cuestión, *después* es verdadero y el ascensor no puede cambiar de dirección. Por lo tanto, debe progresar hasta que alcance el piso.
  - Si este movimiento lo aleja del piso en cuestión, *antes* es verdadero. El ascensor debe en algún momento cambiar de dirección, ya que sólo hay finitos pisos delante, y *antes* será verdadero hasta que cambie de dirección. Cuando lo hace, como *antes* era verdadero, *después* pasa a ser verdadero, y ya se ha establecido que eventualmente llegará al piso en cuestión.

Nótese que este análisis supone la relación correcta entre los botones y los valores *aquí*, *antes* y *después* para cualquier piso. Más adelante se formalizarán, y por ahora se asume que significan lo que se dice que significan.

Ahora se formula la especificación inicial *A_ASCENSOR0*, que se muestra en el Cuadro A.2.

---

```
scheme A_ASCENSOR0 =
  hide movimiento, estado_puerta, piso, direccion, seguro in
  class
    type Ascensor
    value
      /* generadores */
      proximo : T.Requerimiento × Ascensor ~→ Ascensor,
      chequear_botones : Ascensor → T.Requerimiento × Ascensor,
      /* observadores */
      movimiento : Ascensor → T.Movimiento,
      estado_puerta : Ascensor → T.Piso → T.Estado_puerta,
      piso : Ascensor → T.Piso,
      direccion : Ascensor → T.Direccion,
      /* derivado */
      seguro : Ascensor → Bool
      seguro(s) ≡
        (∀ f : T.Piso •
          (estado_puerta(s)(f) = T.abierta) =
          (movimiento(s) = T.parado ∧ piso(s) = f))
    axiom
      [seguro_y_util]
        ∀ s : Ascensor •
          seguro(s) ⇒
          let (r, s') = chequear_botones(s) in
            seguro(s') ∧
            let s" = proximo(r, s') in
              seguro(s") ∧
              (movimiento(s") = T.parado ⇒
```

(T.aqui(r) ∨ (∼ T.despues(r) ∧ ∼ T.antes(r))) ∧
piso(s) = piso(s")) ∧
(movimiento(s") = T.moviendo ⇒
(T.despues(r) ∨ T.antes(r)) ∧
T.es_prox_piso(direccion(s"), piso(s)) ∧
piso(s") = T.prox_piso(direccion(s"), piso(s))) ∧
(direccion(s) ≠ direccion(s") ⇒ ∼ T.despues(r))
**end**
**end**
**end**

Cuadro A.2: Especificación Inicial del Sistema

### Validación

Se debe comprobar que todos los requerimientos estén reflejados en la especificación inicial o atendidos en el plan de desarrollo. En este sistema, las cosas más importantes a comprobar son las propiedades de seguridad y existencia; es necesario controlar que la definición de *es_seguro* y el axioma *seguro_y_util* sean adecuados.

## A.3. Desarrollo del algoritmo principal

El primer objetivo es definir la función *proximo* y mostrar que satisface *seguro_y_util*. Para esto se introducen dos nuevos generadores *mover* y *detener* y se define *proximo* en función de ellos. Por lo tanto *proximo* pasa de ser un generador a ser una función derivada.

*mover* cambia el piso actual al siguiente. Para evitar tener funciones separadas para mover hacia arriba y hacia abajo, tiene un parámetro de dirección, y cambia la dirección actual al valor de su parámetro. También tiene un parámetro de movimiento, ya que si el ascensor ya se está moviendo, *mover* no necesita cerrar ninguna puerta, pero de otra manera la puerta del piso actual debe estar cerrada antes de moverse. *detener* detiene el ascensor en el piso actual y abre (o destraba) las puertas. (Aún es necesario especificar más adelante que libera los botones para ese piso.)

Las precondiciones para *mover* y *detener* son *es_seguro* y, para *mover*, que haya un próximo piso. Es estándar hacer del predicado de seguridad o invariante una precondición para los generadores, ya que la estrategia es mostrar que ésta es preservada por *proximo*, que está definido en términos de *mover* y *detener*. Esto permite calcular efectivamente la precondición para *proximo* desde su cuerpo, notando que sólo llama a *mover* en la dirección original cuando *después* es verdadero, y en la dirección opuesta cuando es verdadero *antes*. Esto además permite calcular una postcondición para *chequear_botones*. La construcción de *A_ASCENSOR1*, que se muestra en el Cuadro A.3 está por lo demás de acuerdo al método para módulos aplicativos abstractos.

---

```
scheme A_ASCENSOR1 =
  hide movimiento, estado_puerta, piso, direccion, mover, detener, seguro in
  class
    type Ascensor
    value
      /* generadores */
      mover : T.Direccion × T.Movimiento × Ascensor ~→ Ascensor,
      parar : Ascensor → Ascensor,
      chequear_botones : Ascensor → T.Requerimiento × Ascensor,
      /* observadores */
      movimiento : Ascensor → T.Movimiento,
      estado_puerta : Ascensor → T.Piso → T.Estado_puerta,
      piso : Ascensor → T.Piso,
      direccion : Ascensor → T.Direccion,
      /* derivado */
      proximo : T.Requerimiento × Ascensor ~→ Ascensor,
      proximo(r, s) ≡
        let d = direccion(s) in
          case movimiento(s) of
            T.parado →
              case r of
                T.mk_Requerim(__, true, __) → mover(d, T.parado, s),
                T.mk_Requerim(__, __, true) →
                  mover(T.invertir(d), T.parado, s),
```

```
                _ _ → s
            end
        T.moviendo →
            case r of
                T.mk_Requerim(true, _, _) → parar(s),
                T.mk_Requerim(_, false, false) → parar(s),
                T.mk_Requerim(_, true, _) → mover(d, T.moviendo, s),
                T.mk_Requerim(_, _, true) →
                    mover(T.invertir(d), T.moviendo, s)
            end
    end
    pre
        (T.despues(r) ⇒ T.es_prox_piso(direccion(s), piso(s))) ∧
        (T.antes(r) ⇒ T.es_prox_piso(T.invertir(direccion(s)), piso(s))),
seguro : Ascensor → Bool
seguro(s) ≡
    (∀ f : T.Piso •
        (estado_puerta(s)(f) = T.abierta) =
        (movimiento(s) = T.parado ∧ piso(s) = f))
axiom
[movimiento_mover]
    ∀ s : Ascensor, d : T.Direccion, m : T.Movimiento •
        movimiento(mover(d, m, s)) ≡ T.moviendo
            pre T.es_prox_piso(d, piso(s)),
[estado_puerta_mover]
    ∀ s : Ascensor, d : T.Direccion, m : T.Movimiento, f : T.Piso •
        estado_puerta(mover(d, m, s))(f) ≡
            if m = T.parado ∧ piso(s) = f then T.cerrada
            else estado_puerta(s)(f) end
            pre T.es_prox_piso(d, piso(s)),
[piso_mover]
    ∀ s : Ascensor, d : T.Direccion, m : T.Movimiento •
```

piso(mover(d, m, s)) ≡ T.prox_piso(d, piso(s))
**pre** T.es_prox_piso(d, piso(s)),
[direccion_mover]
∀ s : Ascensor, d : T.Direccion, m : T.Movimiento •
direccion(mover(d, m, s)) ≡ d **pre** T.es_prox_piso(d, piso(s)),
[mover_definido]
∀ s : Ascensor, d : T.Direccion, m : T.Movimiento •
mover(d, m, s) **post true pre** T.es_prox_piso(d, piso(s)),
[movimiento_parar] ∀ s : Ascensor • movimiento(parar(s)) ≡ T.parado,
[estado_puerta_parar]
∀ s : Ascensor, f : T.Piso •
estado_puerta(parar(s))(f) ≡
**if** piso(s) = f **then** T.abierta **else** estado_puerta(s)(f) **end**,
[piso_parar] ∀ s : Ascensor • piso(parar(s)) ≡ piso(s),
[direccion_parar]
∀ s : Ascensor • direccion(parar(s)) ≡ direccion(s),
[chequear_botones_ax]
∀ s : Ascensor •
chequear_botones(s) **as** (r, s')
**post**
movimiento(s') = movimiento(s) ∧
estado_puerta(s') = estado_puerta(s) ∧
piso(s') = piso(s) ∧
direccion(s') = direccion(s) ∧
(T.despues(r) ⇒ T.es_prox_piso(direccion(s'), piso(s'))) ∧
(T.antes(r) ⇒ T.es_prox_piso(T.invertir(direccion(s')), piso(s')))
**end**

Cuadro A.3: Esquema *A_ASCENSOR1*

## Verificación

La afirmación de que *A_ASCENSOR1* implementa *A_ASCENSOR0* se expresa formulando esta afirmación como una relación de desarrollo y justificándola. Esta

justificación consiste mayormente en justificar que el axioma *seguro_y_util* es verdadero en *A_ASCENSOR1*, es decir que el algoritmo resultará en un ascensor seguro y útil.

## A.4. Descomposición del estado

Se decide modelar el sistema en términos de tres sub-sistemas: las puertas (Cuadro A.4), los botones (Cuadro A.5) y el motor (Cuadro A.6). Cada uno tiene un estado abstracto y funciones que actúan sobre este estado con las cuales se pueden descomponer las acciones de los generadores del modulo *A_ASCENSOR1*.

```
scheme A_PUERTAS0 =
  class
    type Puertas
    value
      /* generadores */
      abrir : T.Piso × Puertas → Puertas,
      cerrar : T.Piso × Puertas → Puertas,
      /* observadores */
      estado_puerta : Puertas → T.Piso → T.Estado_puerta
    axiom
      [estado_puerta_abrir]
        ∀ f, f' : T.Piso, s : Puertas •
          estado_puerta(abrir(f, s))(f') ≡
            if f = f' then T.abierta else estado_puerta(s)(f') end,
      [estado_puerta_cerrar]
        ∀ f, f' : T.Piso, s : Puertas •
          estado_puerta(cerrar(f, s))(f') ≡
            if f = f' then T.cerrada else estado_puerta(s)(f') end
  end
```

Cuadro A.4: Especificación Inicial para las Puertas

```
scheme A_BOTONES0 =
  class
```

**type** Botones
**value**
/* generadores */
liberar : T.Piso × Botones → Botones,
chequear : T.Direccion × T.Piso × Botones →
T.Requerimiento × Botones
**axiom**
[chequear_resultado]
∀ s : Botones, d : T.Direccion, f : T.Piso •
chequear(d, f, s) **as** (r, s')
**post**
(T.despues(r) ⇒ T.es_prox_piso(d, f)) ∧
(T.antes(r) ⇒ T.es_prox_piso(T.invertir(d), f))
**end**

Cuadro A.5: Especificación Inicial para los Botones

---

**scheme** A_MOTOR0 =
**class**
**type** Motor
**value**
/* generadores */
mover : T.Direccion × Motor $\xrightarrow{\sim}$ Motor,
parar : Motor → Motor,
/* observadores */
direccion : Motor → T.Direccion,
movimiento : Motor → T.Movimiento,
piso : Motor → T.Piso
**axiom**
[direccion_mover]
∀ s : Motor, d : T.Direccion •
direccion(mover(d, s)) ≡ d **pre** T.es_prox_piso(d, piso(s)),
[movimiento_mover]

∀ s : Motor, d : T.Direccion •
movimiento(mover(d, s)) ≡ T.moviendo
**pre** T.es_prox_piso(d, piso(s)),
[piso_mover]
∀ s : Motor, d : T.Direccion •
piso(mover(d, s)) ≡ T.prox_piso(d, piso(s))
**pre** T.es_prox_piso(d, piso(s)),
[mover_definido]
∀ s : Motor, d : T.Direccion •
mover(d, s) **post true pre** T.es_prox_piso(d, piso(s)),
[direccion_parar] ∀ s : Motor • direccion(parar(s)) ≡ direccion(s),
[movimiento_parar] ∀ s : Motor • movimiento(parar(s)) ≡ T.parado,
[piso_parar] ∀ s : Motor • piso(parar(s)) ≡ piso(s)
**end**

Cuadro A.6: Especificación Inicial para el Motor

Para formular *A_ASCENSOR2* se usa la siguiente definición concreta para el tipo *Ascensor*:

**Type** Ascensor = M.Motor × DS.Puertas × BS.Botones

Además, se decide eliminar las funciones ocultas *movimiento*, *estado_puerta*, *piso* y *direccion* ya que están ocultas y tienen definiciones simples en términos de funciones correspondientes de los objetos componentes (permitiendo desplegar fácilmente sus ocurrencias). A su vez, esto hace que sea sensato definir *A_ASCENSOR2* en dos etapas, usando un módulo "CUERPO", como lo muestra el Cuadro A.7.

**scheme** A_ASCENSOR2 =
**hide** M, DS, BS, mover, detener, seguro **in** A_ASCENSOR2_CUERPO

**scheme** A_ASCENSOR2_CUERPO =
**class**
**object**
/* motor */

```
  M : A_MOTOR0,
  /* puertas */
  DS : A_PUERTAS0,
  /* botones */
  BS : A_BOTONES0
Type Ascensor = M.Motor × DS.Puertas × BS.Botones
value
  /* generadores */
  mover : T.Direccion × T.Movimiento × Ascensor ~→ Ascensor
  mover(d, m, (ms, ds, bs)) ≡
    (M.mover(d, ms),
    if m = T.parado then DS.cerrar(M.Piso(ms), ds) else ds end,
    bs)
    pre T.es_prox_piso(d, M.piso(ms)),
  parar : Ascensor → Ascensor
  parar((ms, ds, bs)) ≡
    (M.parar(ms), DS.abrir(M.Piso(ms), ds), BS.liberar(M.piso(ms), bs)),
  chequear_botones : Ascensor → T.Requerimiento × Ascensor
  chequear_botones((ms, ds, bs)) ≡
    let (r, bs') = BS.chequear(M.direccion(ms), M.piso(ms), bs) in
      (r, (ms, ds, bs'))
    end
  /* derivado */
  proximo : T.Requerimiento × Ascensor ~→ Ascensor
  proximo(r, (ms, ds, bs)) ≡
    let d = M.direccion(ms) in
      case M.movimiento(ms) of
        T.parado →
          case r of
            T.mk_Requerim(__, true, __) →
              mover(d, T.parado, (ms, ds, bs)),
            T.mk_Requerim(__, __, true) →
              mover(T.invertir(d), T.parado, (ms, ds, bs)),
```

```
                    _ _ → (ms, ds, bs)
                end
            T.moviendo →
                case r of
                    T.mk_Requerim(true, _ _, _ _) → parar((ms, ds, bs)),
                    T.mk_Requerim(_ _, false, false) → parar((ms, ds, bs)),
                    T.mk_Requerim(_ _, true, _ _) →
                        mover(d, T.moviendo, (ms, ds, bs)),
                    T.mk_Requerim(_ _, _ _, true) →
                        mover(T.invertir(d), T.moviendo, (ms, ds, bs))
                end
        end
    end
    pre
        (T.despues(r) ⇒ T.es_prox_piso(M.direccion(ms), M.piso(ms))) ∧
        (T.antes(r) ⇒
            T.es_prox_piso(T.invertir(M.direccion(ms)), M.piso(ms))),
seguro : Ascensor → Bool
seguro((ms, ds, bs)) ≡
    (∀ f : T.Piso •
        (DS.estado_puerta(ds)(f) = T.abierta) =
        (M.movimiento(ms) = T.parado ∧ M.piso(ms) = f))
end
```

Cuadro A.7: Definición de *A_ASCENSOR2*

## Verificación

Se podría querer establecer y justificar una relación de desarrollo entre *A_ASCENSOR1* y *A_ASCENSOR2*. Se podría establecer como

$$\text{A\_ASCENSOR2} \preceq \text{A\_ASCENSOR1}$$

pero esta relación no puede justificarse ya que *A_ASCENSOR1* define y oculta entidades (*movimiento* y otras tres funciones) que no están definidas en *A_ASCENSOR2*.

Claramente no se necesitan, ya que *A_ASCENSOR2* define todas las entidades no ocultas de *A_ASCENSOR1*. Lo que se hace en cambio es mostrar que una extensión de *A_ASCENSOR2_CUERPO* (donde la extensión define *movimiento* y las otras tres funciones) implementa *A_ASCENSOR1*. Se establece la relación en la relación de desarrollo *A_ASCENSOR1_2*, que se muestra en el Cuadro A.8.

**development_relation** [A_ASCENSOR1_2]
  **extend** A_ASCENSOR2_CUERPO **with**
    **class**
      **value**
        movimiento : Ascensor → T.Movimiento
        movimiento((ms, ds, bs)) ≡ M.movimiento(ms),

        estado_puerta : Ascensor → T.Piso → T.Estado_puerta
        estado_puerta((ms, ds, bs)) ≡ DS.estado_puerta(ds),

        piso : Ascensor → T.Piso
        piso((ms, ds, bs)) ≡ M.piso(ms),

        direccion : Ascensor → T.Direccion
        direccion((ms, ds, bs)) ≡ M.direccion(ms)
    **end**
  ⪯ A_ASCENSOR1

Cuadro A.8: Relación de Desarrollo *A_ASCENSOR1_2*

Si hay herramientas disponibles, éstas comprobarán que esta relación sea bien formada, es decir, que se haya incluido todo y no se haya cambiado ninguna signatura. Luego se puede justificar esta relación, que muestra que los axiomas de *A_ASCENSOR1* se cumplen en *A_ASCENSOR2*.

Nótense los siguientes puntos:

- Fue necesario dividir *A_ASCENSOR2* en un "CUERPO" y una parte oculta para poder construir esta relación ya que la extensión que define *movimiento*, etc. debe ser capaz de mencionar nombres como *M* ocultos en *A_ASCENSOR2*.

- La extensión que agrega las entidades ocultas contiene sólo definiciones explícitas, y por lo tanto es improbable que sea inconsistente. Además, por la misma razón, sólo extenderá *A_ASCENSOR2* en forma conservativa.

- Si se puede asegurar que la extensión extiende conservativamente a *A_ASCENSOR2*, se puede asegurar de que las propiedades de las entidades definidas en *A_ASCENSOR2* no son afectadas por la extensión. La extensión usada aquí es realmente conservativa.

- *A_ASCENSOR1* incluye ocultamiento, pero esto puede ignorarse en la justificación ya que todos los nombres ocultos ahora están definidos en la extensión *A_ASCENSOR2_CUERPO*.

Se puede justificar la relación de desarrollo *A_ASCENSOR1_2* para mostrar que *A_ASCENSOR2* implementa *A_ASCENSOR1*. Como la implementación es transitiva, esto muestra que *A_ASCENSOR2* implementa *A_ASCENSOR0*, y en particular que el diseño descompuesto sigue siento seguro y útil.

## A.5. Desarrollo de componentes

Antes de hacer el cambio hacia un sistema concurrente, se construirán los componentes aplicativos concretos definiendo tipos apropiados para sus tipos de interés. En resumen:

- Elegir un tipo RSL concreto para el tipo de interés. Un tipo posible a usar es el producto de los tipos de resultado de los observadores no derivados, pero también se pueden hacer otras elecciones. El criterio importante es que todos los observadores se puedan definir en términos del tipo concreto.

- Proveer cuerpos explícitos para las funciones, usando las características disponibles de RSL para el tipo concreto.

### Motor

Se necesita un tipo concreto para *Motor*. Una elección obvia es

**type** Motor = T.Direccion × T.Movimiento × T. Piso

ya que hay tres observadores en *A_MOTOR0* y cada uno da uno de los tipos del producto. El resto de la formulación de *A_MOTOR1* es simple, y se muestra en el Cuadro A.9.

```
scheme A_MOTOR1 =
  class
    type Motor = T.Direccion × T.Movimiento × T. Piso
    value
      /* generadores */
      mover : T.Direccion × Motor ~→ Motor
      mover(d', (d, m, f)) ≡
        (d', T.moviendo, T.prox_piso(d', f))
        pre T.es_prox_piso(d', f),
      parar : Motor → Motor
      parar((d, m, f)) ≡ (d, T.parado, f),
      /* observadores */
      direccion : Motor → T.Direccion
      direccion((d, m, f)) ≡ d,
      movimiento : Motor → T.Movimiento
      movimiento((d, m, f)) ≡ m,
      piso : Motor → T.Piso
      piso((d, m, f)) ≡ f
  end
```

Cuadro A.9: Formulación de *A_MOTOR1*

## Puertas

En *A_PUERTAS0* sólo hay un observador, con signatura

estado_puerta : Puertas → T.Piso → T.Estado_puerta

Esto sugiere la definición del tipo concreto para el tipo de interés *Puertas*

**type** Puertas = T. Piso → T.Estado_puerta

Puede parecer extraño considerar un tipo de función como suficientemente concreto (ya que los tipos de función generalmente no están disponibles en los lenguajes de programación), pero cuando el tipo del parámetro de esta función es un tipo finito, se puede desarrollar como un arreglo de objetos. Es decir, se tendrá un módulo puerta individual para cada piso. Esto da *A_PUERTAS1*, que se muestra en el Cuadro A.10.

```
scheme A_PUERTAS1 =
  class
    type Puertas = T. Piso → T.Estado_puerta
    value
      /* generadores */
      abrir : T.Piso × Puertas → T. Piso → T.Estado_puerta
      abrir(f, s)(f') ≡ if f = f' then T.abierta else s(f') end,
      cerrar : T.Piso × Puertas → T. Piso → T.Estado_puerta
      cerrar(f, s)(f') ≡ if f = f' then T.cerrada else s(f') end,
      /* observador */
      estado_puerta : Puertas → T.Piso → T.Estado_puerta
      estado_puerta(s) ≡ s
  end
```

Cuadro A.10: Formulación de *A_PUERTAS1*

## Botones

Todavía no se tienen observadores, por lo que no hay una guía clara de qué deberían ser los tipos concretos. También hay que recordar que aún no se modeló la función del usuario de presionar un botón.

Se ha asumido que el ascensor tiene un botón por cada piso, y que cada piso tiene (como máximo) un botón "arriba" y un botón "abajo". El piso inferior sólo tiene un botón "arriba"; el piso superior sólo tiene un botón "abajo"; los pisos intermedios tienen ambos botones.

Hay que recordar que para las puertas, donde se espera tener un arreglo, el tipo concreto era una función con tipo de parámetro *Piso* y tipo de resultado *Estado_puerta*. Los botones pueden modelarse con tres de esos arreglos: botones del as-

censor, botones “arriba” en los pisos, y botones “abajo” en los pisos. Esto sugiere un producto de tipos de función para el tipo concreto, y se puede formular *A_BOTONES1* como se muestra en el Cuadro A.11.

---

**scheme** A_BOTONES1 =
**hide** requerido_aqui, requerido_mas_alla **in**
**class**
**type**
Botones =
(T.Piso → T.Estado_boton) ×
(T.Piso_inf → T.Estado_boton) ×
(T.Piso_sup → T.Estado_boton)
**value**
/* generadores */
liberar : T.Piso × Botones → Botones
liberar(f, (l, u, d)) ≡
(λ f’ : T.Piso • **if** f = f’ **then** T.liberar **else** l(f’) **end**,
λ f’ : T.Piso_inf • **if** f = f’ **then** T.liberar **else** u(f’) **end**,
λ f’ : T.Piso_sup • **if** f = f’ **then** T.liberar **else** d(f’) **end**,
chequear : T.Direccion × T.Piso × Botones →
T.Requerimiento × Botones,
/* observadores */
requerido_aqui : T.Direccion × T.Piso × Botones → **Bool**
requerido_aqui(d, f, (ascensor, arriba, abajo)) ≡
ascensor(f) = T.iluminado ∨
d = T.arriba ∧
(f < T.max_piso ∧ arriba(f) = T.iluminado ∨
f > T.min_piso ∧
abajo(f) =
T.iluminado ∧
∼ requerido_mas_alla(d, f, (ascensor, arriba, abajo))) ∨
d = T.abajo ∧
(f > T.min_piso ∧ abajo(f) = T.iluminado ∨
f > T.max_piso ∧

```
        arriba(f) =
            T.iluminado ∧
                ∼ requerido_mas_alla(d, f, (ascensor, arriba, abajo))),
    requerido_mas_alla : T.Direccion × T.Piso × Botones → Bool
    requerido_mas_alla(d, f, s) ≡
        T.es_prox_piso(d, f) ∧
        let f' = T.prox_piso(d, f) in
            requerido_aqui(d, f', s) ∨ requerido_mas_alla(d, f', s)
        end
axiom
    [chequear_resultado]
        ∀ s : Botones, d : T.Direccion, f : T.Piso •
    chequear(d, f, s) as (r, s')
        post
            r =
            T.mk_Requerim
                (requerido_aqui(d, f, s),
                requerido_mas_alla(d, f, s),
                requerido_mas_alla(T.invertir(d), f, s))
end
```

Cuadro A.11: Formulación de *A_BOTONES1*

Nótese que por primera vez se pudo definir qué hace *liberar* y cómo se calculan *aquí*, *antes* y *después*.

## Verificación

Para cada uno de los módulos aplicativos, motor, puertas, y botones, es fácil formular y justificar que la versión concreta implementa la versión abstracta.

## Validación

Como se ha elaborado por primera vez *liberar* y *aquí*, *antes* y *después* para el módulo de botones, es necesario comprobar que estos son lo que se requería. Por

ejemplo, *liberar* liberará el botón hacia abajo para un piso si el ascensor se detiene allí en su camino ascendente (debido a alguna otra llamada). Se asume que en esta situación lo más probable es que cualquier persona que esté esperando para bajar entre al ascensor, y que por lo tanto volver a detenerse en ese piso cuando el ascensor está descendiendo será claramente una pérdida de tiempo. Esto puede ser apropiado para un sistema de un único ascensor, pero no para un sistema con múltiples ascensores. Otra alternativa es que el ascensor se detenga cuando va hacia arriba sólo si no hay ningún requerimiento para ir más arriba. Sin duda hay otras alternativas, quizás mejores, pero el propósito principal de este ejemplo es mostrar cómo se pueden especificar y desarrollar estos sistemas, y no discutir en detalle algoritmos de control para ascensores.

## A.6. Introducción de concurrencia

### Ascensor

Hay cuatro módulos aplicativos concretos para transformar en concurrentes. El más fácil es el módulo de composición *A_ASCENSOR2*, que se transformará en *C_ASCENSOR2*. En resumen, lo que se hace es:

- Definir objetos *M*, *DS* y *BS* como en la versión aplicativa, pero esta vez instanciando versiones imperativas concurrentes de los módulos de motor, puertas y botones. No hay definición del tipo *Ascensor*.
- Para cada una de las funciones, incluir en su tipo el acceso **in any out any** y borrar el tipo de interés *Ascensor* de sus tipos de parámetros y resultado (reemplazando por **Unit** si no hay otros componentes en un tipo de parámetros o resultado).
- Definir los cuerpos de las funciones adaptando las versiones aplicativas para usar las funciones imperativas correspondientes a las aplicativas.
- Agregar una función *init* para llamar a todas las funciones *init* de los objetos integrantes en paralelo.

Esto resulta en *C_ASCENSOR2*, cuya formulación se muestra en el Cuadro A.12.

```
scheme C_ASCENSOR2 =
  hide M, DS, BS, mover, detener in
  class
    object
      /* motor */
      M : C_MOTOR1,
      /* puertas */
      DS : C_PUERTAS1,
      /* botones */
      BS : C_BOTONES1
    value
      /* generadores */
      mover : T.Direccion × T.Movimiento → in any out any Unit
      mover(d, m) ≡
        if m = T.parado then DS.cerrar(M.Piso()) end ; M.mover(d),
      parar : Unit → in any out any Unit
      parar() ≡
        let f = M.piso() in BS.liberar(f) ; M.parar() ; DS.abrir(f) end,
      chequear_botones : Unit → in any out any T.Requerimiento
      chequear_botones() ≡ BS.chequear(M.direccion(), M.piso()),
      /* derivado */
      proximo : T.Requerimiento → in any out any Unit
      proximo(r) ≡
        let d = M.direccion() in
          case M.movimiento() of
            T.parado →
              case r of
                T.mk_Requerim(__, true, __) → mover(d, T.parado),
                T.mk_Requerim(__, __, true) →
                  mover(T.invertir(d), T.parado),
                __ → skip
              end
```

```
                T.moviendo →
                  case r of
                    T.mk_Requerim(true, _, _) → parar(),
                    T.mk_Requerim(_, false, false) → parar(),
                    T.mk_Requerim(_, true, _) → mover(d, T.moviendo),
                    T.mk_Requerim(_, _, true) →
                      mover(T.invertir(d), T.moviendo)
                  end
              end
            end
        /* inicial */
        init : Unit → in any out any write any Unit
        init() ≡ M.init() ‖ DS.init() ‖ BS.init(),
        /* control */
        ascensor : Unit → in any out any Unit
        ascensor() ≡ while true do proximo(chequear_botones()) end
    end
```

Cuadro A.12: Formulación de *C_ASCENSOR2*

Aún no se han formulado los módulos components *C_MOTOR1*, etc., pero el método es lo bastante regular como para escribir *C_ASCENSOR2* incluso cuando todavía no se pueden chequear sus tipos. También se incluyó la función de control *ascensor* que sigue el patrón que se indicó anteriormente: comprueba repetidamente los botones y realiza la próxima acción. ¿Por qué no se escribió la contraparte de esta función en la versión aplicativa *A_ASCENSOR2* o en alguna anterior?

Si se hubiese tratado de escribir esta función en la versión aplicativa, se hubiese podido escribir algo como lo siguiente:

```
value
    ascensor : Ascensor → Ascensor
    ascensor(s) ≡ ascensor(proximo(chequear_botones(s)))
```

Pero esta definición probablemente sería contradictoria. Se afirma que la función *ascensor* es convergente (por la flecha de función total en su tipo) cuando se aplica y por lo tanto (ya que es aplicativa) debe terminar cuando se aplica. Pero tal función ge-

neralmente no terminará. La contraparte concurrente es convergente porque, aunque involucra un ciclo infinito, este ciclo comunica. Por lo que se resalta nuevamente que la forma de especificar y analizar tales sistemas comenzando con una especificación aplicativa es en términos de una función "próximo".

## Motor

El motor es el más fácil de los tres componentes porque no involucra arreglos de componentes. En resumen, lo que se hace es:

- Definir una variable para cada componente del tipo aplicativo de interés de *A_MOTOR1*.
- Dar signaturas a las funciones correspondientes a las aplicativas mediante el agregado de los accesos **in any out any** y la eliminación del tipo de interés *Motor* (como es usual, agregando **Unit** donde sea necesario).
- Definir canales para (al menos) los tipos de parámetro y resultado de las funciones que no son **Unit**.
- Definir el cuerpo de cada función como una salida de su parámetro (excepto las de tipo **Unit** sin canales) seguida por una entrada de su resultado (excepto las de tipo **Unit** sin canales).
- Agregar una función "principal" (aquí llamada *motor*) que es un ciclo **while true do** que contiene una elección externa entre una expresión para cada una de las otras funciones. Cada una de estas expresiones
  - ingresa el valor del parámetro desde la función (si existe), luego
  - para los generadores, actualiza las variables como sea apropiado, luego
  - devuelve el valor del resultado a la función (si existe).

  La actualización de las variables y el resultado devuelto son las contrapartes imperativas de los cuerpos de las funciones aplicativas.
- Definir una función *init* que llama a la función principal luego de, posiblemente, inicializar las variables.

La formulación de *C_MOTOR1* se muestra en el Cuadro A.13.

```
scheme C_MOTOR1 =
  hide CH, V, motor in
  class
    object
      CH :
        class
          channel
            direccion : T.Direccion,
            piso : T.Piso,
            movimiento : T.Movimiento,
            mover : T.Direccion,
            parar, mover_ack, parar_ack : Unit
        end,
      V :
        class
          variable
            direccion : T.Direccion,
            movimiento : T.Movimiento,
            piso : T.Piso
        end
    value
      /* principal */
      motor : Unit → in any out any write any Unit
      motor() ≡
        while true do
          let d' = CH.mover? in
            CH.mover_ack ! () ; V.direccion := d' ;
            V.movimiento := T.moviendo ;
            V.piso := T.prox_piso(d', V.piso)
          end []
          CH.parar? ; CH.parar_ack ! () ; V.movimiento := T.parado []
          CH.direccion ! V.direccion []
```

```
            CH.movimiento ! V.movimiento []
            CH.piso ! V.piso
        end,
    /* inicial */
    init : Unit → in any out any write any Unit
    init() ≡ motor(),
    /* generadores */
    /* asume que sólo se llama a mover cuando */
    /* existe un próximo piso en la dirección actual */
    mover : T.Direccion → in any out any Unit
    mover(d) ≡ CH.mover ! d ; CH.mover_ack?,
    parar : Unit → in any out any Unit
    parar() ≡ CH.parar ! () ; CH.parar_ack?,
    /* observadores */
    direccion : Unit → in any out any T.Direccion
    direccion() ≡ CH.direccion?,
    piso : Unit → in any out any T.Piso
    piso() ≡ CH.piso?,
    movimiento : Unit → in any out any T.Movimiento
    movimiento() ≡ CH.movimiento?
end
```

Cuadro A.13: Formulación de *C_MOTOR1*

Aquí hay algunas decisiones de diseño interesantes:

- Como el estado concreto era un producto de tres componentes, es natural usar tres variables.
- Como en el caso de los canales, las variables se pusieron dentro de un objeto $V$ para facilitar su ocultamiento.
- No se dan valores iniciales para las variables. Esto refleja la intuición de que puede ser necesario iniciar o reiniciar el ascensor en cualquier estado. En este módulo se podría querer asegurar que el ascensor está, por ejemplo, estacionado

en el piso inferior y preparado para subir, pero entonces no sería aplicable para un reinicio del sistema en algún otro estado. (Esa facilidad de reinicio puede necesitar algunas funciones iniciales para permitir que el sistema sea seguro antes de comenzar su comportamiento normal.)

- Hay canales de "reconocimiento" para los resultados de las funciones *mover* y *parar*, aunque éstas sean de tipo **Unit**. Esto permite asumir que el motor del ascensor en realidad ha realizado la acción correspondiente cuando la función termina. Nótese que en la especificación no hay nada sobre el envío real de comandos al motor físico. Hay dos formas de interpretar esta especificación (que afectarán la forma en que se traduzca):

  - Se considera la definición de *motor* como una especificación de las *suposiciones* sobre la interfaz de hardware; especifica que luego de un *mover*, por ejemplo, las variables se ajustan de forma que las funciones consecuentes como *piso* obtendrán información que corresponde tanto a qué se supone que mover debe hacer (cambiar la variable de piso al próximo piso) como a qué hizo realmente el ascensor físico. Las funciones *mover*, *parar*, etc. son la interfaz al hardware. En este caso la traducción de los módulos ignorará *motor* y traducirá las funciones en términos de llamadas al hardware.
  - Se considera que *motor* llama implícitamente a las funciones de hardware en los lugares apropiados, como por ejemplo, que le dice al motor real que se mueva luego de recibir una entrada en el canal *CH.mover* y luego que espere un reconocimiento del motor antes de provocar una salida en el canal *CH.mover_ack*. En este caso la función *motor* se traducirá de forma que incluya las llamadas apropiadas al hardware.

  Qué interpretación se tome depende mayormente de cuán cerca está *motor* de la forma en que realmente opera el hardware. En este caso se toma la primera interpretación.
  Nótese que, si se tomara la segunda interpretación, la traducción de *C_ASCENSOR2* también se ve afectada; sería necesario asegurarse de que inicialmente se invoque la función *init*, ya que ésta llamará a los procesos iniciales de motor,

puertas y botones. Con la primera interpretación, todos éstos ya están "corriendo" como hardware.

- La función aplicativa *mover* de *A_MOTOR1* tenía una precondición que incluía un parámetro, por lo que no se podía usar el estilo "si precondición entonces comunicar sino parar". Sin embargo, se puede comprobar que en todas las llamadas de *mover* la precondición (que haya un piso válido hacia el cual moverse) es verdadera, por lo que esta es una instancia donde la comprobación se puede omitir. El chequeo, de hecho, era parte de la prueba del axioma *seguro_y_util* anterior, ya que en esa prueba sólo se podían desarrollar llamadas de *mover* para las cuales la precondición es verdadera. Como esto reduce la robustez de *C_MOTOR1*, se incluyó un comentario sobre esta característica. Una implementación más robusta incluiría algo de código en la parte apropiada de *motor*, que requeriría más reconocimiento sobre el hardware involucrado.

## Puertas

Se necesita un método para descomposición en un arreglo de objetos. Esto es así:

- En el tipo concreto de interés se tiene un componente que es un tipo de función. El tipo de parámetro de este tipo de función será el tipo del índice del arreglo. (Esto se puede hacer en RSL aún si el tipo es infinito, pero típicamente sólo es de utilidad si el tipo es finito y bastante pequeño. De otro modo se necesita reconsiderar el tipo concreto.)

- Se necesita una expresión de clase para el arreglo. Normalmente ésta se define como un esquema separado.

- El tipo de interés de este esquema será el tipo de resultado del tipo de función, en este caso *T.Estado_puerta*.

- Este método sólo tiene sentido si el esquema actual es aplicativo y el desarrollo es hacia uno imperativo o concurrente con un componente imperativo o concurrente.

- Es necesario definir las funciones del esquema componente. Por lo general esto es muy obvio, o pronto lo es, porque son las funciones necesarias para modelar

las (contrapartes imperativas o concurrentes de las) funciones en el módulo actual. Cualquier generador en el módulo actual que cambia o depende de una aplicación de este componente necesitará una o más funciones correspondientes; cualquier observador que produce un valor del tipo o en su cuerpo aplica un valor del tipo necesitará una o más funciones correspondientes.

- Se completa la definición de las funciones del módulo componente y se usan en el desarrollo del módulo actual.
- Para un desarrollo concurrente, la función *init* del módulo actual será definida como la composición paralela de las funciones *init* de los componentes.

Es claro que el módulo componente para una única puerta necesitará funciones *abrir*, *cerrar*, y *estado_puerta*. Se lo formula como el módulo concurrente *C_PUERTA1*, que se muestra en el Cuadro A.14.

---

```
scheme C_PUERTA1 =
  hide CH, puerta_var, puerta in
  class
    object
      CH :
        class
          channel
            abrir, cerrar, abrir_ack, cerrar_ack : Unit,
            estado_puerta : T.Estado_puerta
        end,
      variable puerta_var : T.Estado_puerta
      value
        /* principal */
        puerta : Unit → in any out any write any Unit
        puerta() ≡
          while true do
            CH.abrir? ; CH.abrir_ack ! () ; puerta_var := T.abierta []
            CH.cerrar? ; CH.cerrar_ack ! () ; puerta_var := T.cerrada []
            CH.estado_puerta ! puerta_var
          end,
```

```
        /* inicial */
        init : Unit → in any out any write any Unit
        init() ≡ puerta(),
        /* generadores */
        cerrar : Unit → in any out any Unit
        cerrar() ≡ CH.cerrar ! () ; CH.cerrar_ack?,
        abrir : Unit → in any out any Unit
        abrir() ≡ CH.abrir ! () ; CH.abrir_ack?,
        /* observador */
        estado_puerta : Unit → in any out any T.Estado_puerta
        estado_puerta() ≡ CH.estado_puerta?
    end
```

Cuadro A.14: Formulación de *C_PUERTA1*

Nótese que, como en el caso del módulo motor:

- No se hicieron suposiciones sobre el estado inicial.
- Se incluyeron reconocimientos para poder asumir que las interacciones con el hardware están completas cuando sus funciones terminan.

Esto permite formular *C_PUERTAS1* como se muestra en el Cuadro A.15.

```
scheme C_PUERTAS1 =
    hide DS in
    class
        object DS[f : T.Piso] : C_PUERTA1
        value
            /* inicial */
            init : Unit → in any out any write any Unit
            init() ≡ ‖ { DS[f].init() | f : T.Piso },
            /* generadores */
            abrir : T.Piso → in any out any Unit
            abrir(f) ≡ DS[f].abrir(),
            cerrar : T.Piso → in any out any Unit
```

```
        cerrar(f) ≡ DS[f].cerrar(),
        /* observador */
        estado_puerta : T.Piso → in any out any T.Estado_puerta
        estado_puerta(f) ≡ DS[f].estado_puerta()
end
```

Cuadro A.15: Formulación de *C_PUERTAS1*

## Botones

El método es el mismo que para las puertas, excepto que se necesitan tres arreglos, cada uno del mismo esquema componente *C_BOTON1* que se muestra en el Cuadro A.16. Las transiciones de estado para cada botón se muestran en la Figura A.5.

```
scheme C_BOTON1 =
    hide CH, boton, boton_var in
    class
        object CH : class channel pulsar, liberar : Unit,
            chequear : T.estado_boton end
        variable boton_var : T.Estado_boton
        value
            /* principal */
            boton : Unit → in any out any write any Unit
            boton() ≡
                while true do
                    CH.pulsar? ; boton_var := T.iluminado []
                    CH.liberar? ; boton_var := T.liberado []
                    CH.chequear ! boton_var
                end,
            /* inicial */
            init : Unit → in any out any write any Unit
            init() ≡ boton(),
            /* generadores */
            pulsar : Unit → in any out any Unit
```

pulsar() ≡ CH.pulsar ! (),
liberar : **Unit → in any out any Unit**
liberar() ≡ CH.liberar ! (),
/* observador */
chequear : **Unit → in any out any** T.Estado_boton
chequear() ≡ CH.chequear?

**end**

Cuadro A.16: Formulación del Esquema Componente *C_BOTON1*

Como en el caso del motor y las puertas, inicialmente no se hicieron suposiciones sobre los botones.

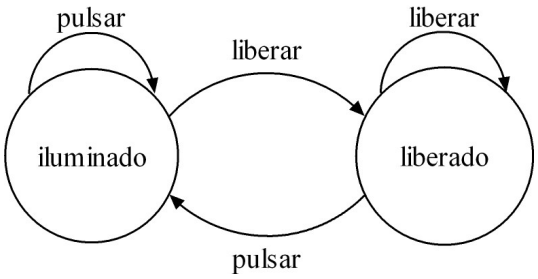


Figura A.5: Transiciones de estado para un botón

El paso de desarrollo de *A_BOTONES1* a *C_BOTONES1* usando arreglos ahora es directo, como se muestra en el Cuadro A.17.

**scheme** C_BOTONES1 =
**hide** LB, UB, DB, requerido_aqui, requerido_mas_alla **in**
**class**
**object**
/* botones del ascensor */
LB[f : T.Piso] : C_BOTON1,
/* botones arriba */
UB[f : T.Piso_inf] : C_BOTON1,
/* botones abajo */
DB[f : T.Piso_sup] : C_BOTON1

**value**

/* inicial */

init : **Unit** → **in any out any write any Unit**

init() ≡

‖ { LB[f].init() | f : T.Piso } ‖

‖ { UB[f].init() | f : T.Piso_inf } ‖

‖ { DB[f].init() | f : T.Piso_sup },

/* generadores */

liberar : T.Piso → **in any out any Unit**

liberar(f) ≡

LB[f].liberar() ;

**if** f < T.max_piso **then** UB[f].liberar() **end** ;

**if** f > T.min_piso **then** DB[f].liberar() **end**,

/* observadores */

chequear : T.Direccion × T.Piso → **in any out any** T.Requerimiento

chequear(d, f) ≡

T.mk_Requerim

(requerido_aqui(d, f),

requerido_mas_alla(d, f),

requerido_mas_alla(T.invertir(d), f)),

requerido_aqui : T.Direccion × T.Piso → **in any out any Bool**

requerido_aqui(d, f) ≡

LB[f].chequear() = T.iluminado ∨

d = T.arriba ∧

(f < T.max_piso ∧ UB[f].chequear() = T.iluminado ∨

f > T.min_piso ∧

DB[f].chequear() = T.iluminado ∧ ∼ requerido_mas_alla(d, f)) ∨

d = T.abajo ∧

(f > T.min_piso ∧ DB[f].chequear() = T.iluminado ∨

f < T.max_piso ∧

UB[f].chequear() = T.iluminado ∧ ∼ requerido_mas_alla(d, f)),

requerido_mas_alla : T.Direccion × T.Piso → **in any out any Bool**

requerido_mas_alla(d, f) ≡

```
            T.es_prox_piso(d, f) ∧
            let f' = T.prox_piso(d, f) in
               requerido_aqui(d, f') ∨ requerido_mas_alla(d, f')
            end
      end
```

Cuadro A.17: Paso de Desarrollo de *A_BOTONES1* a *C_BOTONES1*

## Verificación

Como este paso de desarrollo fue de aplicativo a concurrente, es necesario decidir qué nivel de seguridad se necesita para la correctitud. Las opciones son:

- chequear que el método para esta transición se siguió correctamente, o
- formular el axioma concurrente correspondiente al axioma aplicativo *seguro_y_util* de *A_ASCENSOR0* y justificarlo para *C_ASCENSOR2*.

Ambas son verificaciones, ya que comprueban sobre la correctitud del proceso de desarrollo. La primera es informal y generalmente es todo lo que se necesita. La segunda es formal y puede hacerse si se tiene alguna duda o si se requiere el mayor nivel de garantía de correctitud.

# Bibliografía


[Amb95] S. Ambler. Using use clases. *Software Development*, pages 53–61, jul 1995.

[ASM80] J. R. Abrial, S. A. Schuman, and B. Meyer. Specification language. In R. M. McKeag and A. M. Macnaghten, editors, *On the Construction of Programs: An Advanced Course*, pages 343–410. Cambridge University Press, 1980.

[B+98] Barry Boehm et al. Using the winwin spiral model: A case study. *Computer*, 31(7):33–44, July 1998.

[BB94] Barry Boehm and Prasanta Bose. A collaborative spiral software process model based on theory W. In IEEE CS Press, editor, *Applying the Software Process*, pages 59–68, August 1994.

[BCK99] L. Bass, P. Clements, and R. Kazman. *Software Arquitecture in Practice.* Addison-Wesley, sixth edition, 1999.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition.* Addison-Wesley Professional, April 2003.

[BeAC94] Fernando Brito e Abreu and Rogério Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1), jul 1994.

[BeAM96] Fernando Brito e Abreu and Walcelio Melo. Evaluating the impact of object-oriented design on software quality. In *METRICS '96: Proceedings of the 3rd International Symposium on Software Metrics*, page 90, Washington, DC, USA, 1996. IEEE Computer Society.

[Ber93] E. V. Berard. *Essays on Object-Oriented Software Engineering*. Addison-Wesley, 1993.

[Ber95] E. Berard. Metrics for object-oriented software engineering. Publicado en Internet en comp.software-eng, jan 1995.

[BG80] Rod M. Burstall and Joseph A. Goguen. The semantics of clear, a specification language. In *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*, pages 292–332, London, UK, 1980. Springer-Verlag.

[BG81] Rod M. Burstall and Joseph A. Goguen. An informal introduction to specification using clear. In R. Boyer and J. Stothers Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic Press, 1981.

[BH95a] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, jul 1995.

[BH95b] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.

[Big00] Peter Biggs. A survey of object-oriented methods. Universidad de Durham, http://students.cs.byu.edu/~pbiggs/survey.html, 2000.

[Bil91] S. Bilow. Book review: Object-oriented design. *Journal of Object Oriented Programming*, 4(6):73–74, oct 1991.

[Bin94a] R. V. Binder. Object-oriented software testing. *Communications of the ACM*, 37(9):29, sep 1994.

[Bin94b] R. V. Binder. Testing oo systems: A status report. *American Programmer*, 7(4):23–28, apr 1994.

[BJ82] D. Bjørner and C. Jones. *Formal Specification and Software Development*. 1982.

[BM88] Robert Boyer and J. Strother Moore. *A Computational Logic Handbook*. Number 23 in Perspectives in Computing. Academic Press, 1988.

[BMS80] Rod M. Burstall, David B. MacQueen, and Donald Sannella. Hope: An experimental applicative language. In *LISP Conference*, pages 136–143, 1980.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[Boe89] Barry W. Boehm. *Software Risk Management*. IEEE Computer Society Press, Washington u.a., 1989.

[Boe96] Barry Boehm. Anchoring the software process. *IEEE Software*, 13(4):73–82, July 1996.

[Boe00a] Barry Boehm. Spiral development: Experience, principles, and refinements. *Spiral Development Workshop*, feb 2000.

[Boe00b] Barry Boehm. Unifying software engineering and systems engineering. *Computer*, 33(3):114–116, March 2000.

[Boe06] Barry Boehm. A view of 20th and 21st century software engineering. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 12–29, New York, NY, USA, 2006. ACM Press.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin-Cummings, 1991.

[BR89] Barry Boehm and Rony Ross. Theory-W software project management: Principles and examples. *IEEE Transactions on Software Engineering*, 15(7):902–916, July 1989.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[Bus93] F. Buschmann. Rational architectures for object-oriented software systems. *Journal of Object-Oriented Programming*, 6(5):30–41, September 1993.

[C+89] Luca Cardelli et al. Modula-3 report. Technical Report SRC-RR-52, Digital System Research Center, 1989.

[C+92] Luca Cardelli et al. Modula-3 report (revised). *ACM SIGPLAN Notices*, 27(8):15–42, 1992.

[CB95] B. Clark and B. Boehm. Knowledge summary. In *Focused Workshop on COCOMO 2.0*, May 1995.

[CK94] S. R. Chidamber and C. F. Kemerer. A metric suite for object-oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, jun 1994.

[CKI88] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11), nov 1988.

[CKV03] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for formal verification. Technical report, Hebrew University – Rice University, nov 2003.

[Cop96] James O. Coplien. *Software Patterns*. SIGS Books & Multimedia, 1996.

[Cox86] B. J. Cox. *Object Oriented Programming*. Addison-Wesley, 1986.

[CS95a] N. I. Churcher and M. J. Shepperd. Towards a conceptual framework for object-oriented metrics. *ACM Software Engineering Notes*, 20(2):69–76, apr 1995.

[CS95b] Michael A. Cusumano and Richard W. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press, New York, NY, USA, 1995.

[CY90] P. Coad and E. Yourdon. *Object Oriented Analysis*. Prentice-Hall, 1990.

[CY91] P. Coad and E. Yourdon. *Object Oriented Design*. Prentice-Hall, 1991.

[DMN67] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA 67 common base language, (Norwegian Computing Center Publication)*. 1967.

[E+01] Vazquez Escudero et al. Métricas orientadas a objetos. Informe técnico, Departamento de Informática y Automática, Facultad de Ciencias, Universidad de Salamanca, nov 2001.

[F+03] José Fuentes et al. Errors in the UML metamodel? *SIGSOFT Software Engineering Notes*, 28(6):3–3, November 2003.

[FG+85] Kokichi Futatsugi, Joseph A. Goguen, et al. Principles of obj2. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 52–66, 1985.

[FK92] R. G. Fichman and C. F. Kemerer. Object-oriented and conventional analysis and design methodologies. *Computer*, 25(10):22–39, oct 1992.

[Fle06] Arthur Fleck. Formal methods in software engineering, página del curso. http://www.cs.uiowa.edu/~fleck/181.html, 2006.

[FMC96] Kevin Forsberg, Hal Mooz, and Howard Cotterman. *Visualizing Project Management: Models and Frameworks for Mastering Complex Systems*. Wiley Publishers, 1996.

[FP96] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thompson Computer Press, 1996.

[G+92] Chris George et al. *The Raise Specification Language*. Prentice Hall, New York, 1992.

[G+93] Erich Gamma et al. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[G+95] Erich Gamma et al. *Design Patterns*. Addison-Wesley, 1995.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[GB80] J. A. Goguen and R. M. Burstall. Cat, a system for the structured elaboration of correct programs from structured specifications. Technical report, sri, international, Computer Science Lab, 1980.

[GH83] J. Guttag and J. Horning. Preliminary report on the larch shared language, 1983.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. 1993.

[Gib94] W. Wayt Gibbs. Software's chronic crisis. *Scientific American*, pages 86–95, sep 1994.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[GM86] J. A. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 295–364. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[GM96] J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, Mass., 1st edition, 1996.

[Gog] Joseph A. Goguen. Hidden algebra for software engineering. Página del Departamento. www-cse.ucsd.edu/users/goguen/ps/dmtcs.ps.gz, Department of Computer Science & Engineering, University of California at San Diego, La Jolla.

[Gos91] S. Gossain. Book review: Designing object-oriented software. *Journal of Object Oriented Programming*, 4(1):82–84, mar/apr 1991.

[GR83a] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[GR83b] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[Gro95] The RAISE Method Group. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall, 1995.

[GTW78] Joseph A. Goguen, James W. Thatcher, and Eric G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology, IV: Data Structuring*, pages 80–149. Prentice-Hall, New Jersey, 1978.

[Gut75] John Vogel Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto, Department of Computer Science, 1975.

[Hal77] Maurice H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[Hal90] J. A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, sep 1990.

[Hal96] Anthony Hall. Using formal methods to develop an atc information system. *IEEE Software*, 13(2):66–76, sep 1996.

[Hal98] Elaine M. Hall. *Managing risk: methods for software systems development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[Har86] Robert Harper. Introduction to standard ml. Technical Report ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, November 1986.

[HCN00] R. Harrison, S. Counsell, and R.Ñithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *The Journal of Systems and Software*, 52(2–3):173–179, 2000.

[HG92] C.A.R. Hoare and M.J.C. Gordon. *Mechanized Reasoning and Hardware Design*. Prentice-Hall, 1992.

[HH96] R.P. Higuera and Y.Y. Haimes. Software risk management. Technical Report CMU/SEI96-TR-012, Software Engineering Insitute, 1996.

[HMM86] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, March 1986.

[Hoa72] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hos00] Chris Hostetter. Survey of object oriented programming languages. http://www.rescomp.berkeley.edu/~hossman/cs263/paper.html, 2000.

[HS96] Brian Henderson-Sellers. *Object-Oriented Metrics, measures of complexity.* Prentice Hall, 1996.

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[Jon80] Cliff B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1980.

[Jon94] Capers Jones. *Assessment and control of software risks.* Yourdon Press, Upper Saddle River, NJ, USA, 1994.

[Kar96] Dale W. Karolak. *Software Engineering Management.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[Kay96] Alan C. Kay. The early history of smalltalk. pages 511–598, 1996.

[Kro93] Klaus Kronlöf, editor. *Method integration: concepts and case studies.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[KT94] S. Kirani and W. T. Tsai. Specification and verification of object-oriented programs. Technical report tr 94-96, Computer Science Department, University of Minnesota, dec 1994.

[Las06] Christopher G. Lasater. *Design Patterns (Wordware Applications Library).* Wordware Publishing Inc., Plano, TX, USA, 2006.

[LG97] Luqi and Joseph A. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, 1997.

[LK94] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, 1994.

[Luq89] Luqi. Software evolution through rapid prototyping. *Computer*, 22(5):13–25, 1989.

[Luq96] Luqi. System engineering and computer-aided prototyping. *Systems Integration, Special Issue on Computer Aided Prototyping*, 6(1):15–17, 1996.

[M+97] Robert T. Monroe et al. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43–52, January 1997.

[MBA06] Rafael Menéndez-Barzanallana Asensio. Curso: Informática aplicada a la gestión pública. Publicado en Internet en http://www.um.es/, dec 2006.

[McC76] T. McCabe. A software complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[MDL87] Harlan D. Mills, Michael Dyer, and Richard C. Linger. *Cleanroom Software Engineering*. IEEE Software, sep 1987.

[Mey90] Bertrand Meyer. *Object Oriented Software Construction*. Prentice-Hall, second edition, 1990.

[Mey91] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Mey97] Bertrand Meyer. UML: The positive spin. Publicado en Internet en http://www.eiffel.com/, 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[MK94] J. D. McGregor and T. D. Korson. Integrated object-oriented testing and development process. *Communications of the ACM*, 37(9):59–77, sep 1994.

[MP92] D. E. Monarchi and G. I. Puhr. A research typology for object-oriented analysis and design. *Communications of the ACM*, 35(9):35–47, sep 1992.

[Nau68] Naur, P. and Randell, B., editor. *Proceedings of the NATO Conference on Software Engineering*, Garmish, Germany, October 1968. NATO Science Committee.

[Neu95] Peter G. Neumann. *Computer related risks*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[Nog00a] Juan C Nogueira. A formal risk assessment model for software projects. Ph.D. Dissertation, Naval Postgraduate School, 2000.

[Nog00b] Juan C. Nogueira. *A Risk Assessment Model for Evolutionary Software Projects*. PhD thesis, Naval Postgraduate School, 2000.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[Pre97] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, fourth edition, 1997.

[PS97] Wolfgang Pree and Hermann Sikora. Design patterns for object-oriented software development. pages 663–664, 1997.

[R+91] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Rob92a] P. J. Robinson. *Hierarchical Object-Oriented Design*. Prentice-Hall, 1992.

[Rob92b] P. J. Robinson. Introduction and overview. *Object-Oriented Design*, pages 1–10, 1992.

[San84] Donald Sannella. A set-theoretic semantics for clear. *Acta Informatica*, 21:443–472, 1984.

[San88] D. Sannella. A survey of formal software development methods. Technical Report ECS–LFCS–88–56, 1988.

[SE04] Tommy Staffans and Andreas Enbacka. Metrics for formal methods: The atc case study. Technical report, Department of Computer Science, Abo Akademi University, dec 2004.

[Sem92] Docker Semmens, France. Integrating structured analysis and formal specification techniques. *The Computer*, pages 600–610, dec 1992.

[SH04] Colin F. Snook and Rachel Harrison. Experimental comparison of the comprehensibility of a z specification and its implementation in java. *Information & Software Technology*, 46(14):955–971, 2004.

[SM92] S. Shlaer and S. Mellor. *Object Lifecycles: Modeling the World in States*. Prentice-Hall, 1992.

[SO96] David Stoutamire and Stephen Omohundro. The sather 1.1 specification. Technical Report TR-96-012, Berkeley, CA, 1996.

[SPHP] Ignacio Serrano Pozo and María Inés Herrero Platero. Análisis comparativo del modelado de sistemas en SDL y UML. Publicado en Internet, Departamento de Ingeniería de Comunicaciones, Universidad de Málaga.

[Spi88] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, New York, NY, USA, 1988.

[Spi89] J. M. Spivey. An introduction to z and formal specifications. *Softw. Eng. J.*, 4(1):40–50, 1989.

[ST86] D Sannella and A Tarlecki. Extended ml: an institution-independent framework for formal program development. In *Proceedings of a tutorial and workshop on Category theory and computer programming*, pages 364–389, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[ST04] Alan Shalloway and James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design (2nd Edition) (Software Patterns Series)*. Addison-Wesley Professional, October 2004.

[Std93] IEEE Std. IEEE Software Engineering Standard: Glossary of Software Engineering Terminology. IEEE Computer Society Press, 1993.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Reading, MA, USA, 1986.

[Tay90] D. A. Taylor. *Object-Oriented Technology: A Manager's Guide*. Addison-Wesley, 1990.

[Tho98] J. Thorp. *The Information Paradox: Realizing the Benefits of Information Technology*. McGraw-Hill, 1998.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[VLK98] Rick Vinter, Martin Loomes, and Diana Kornbrot. Applying software metrics to formal specifications: A cognitive approach. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, page 216, Washington, DC, USA, 1998. IEEE Computer Society.

[Wal92] I. J. Walker. Requirements of an object-oriented design method. *Software Engineering Journal*, 7(2):102–113, mar 1992.

[WB+90] Wirfs-Brock et al. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[Wei51] Waloddi Weibull. A statistical distribution function of wide applicability. *Journal of Applied Mechanics*, 18:293–297, 1951.

[WN95] Kim Waldén and Jean-Marie Nerson. *Seamless object-oriented software architecture: analysis and design of reliable systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[WY04] Fangjun Wu and Tong Yi. Measuring z specifications. *ACM SIGSOFT Software Engineering Notes*, 29(6), Sep 2004.

[Zil74] S. Zilles. Algebraic specification of data types. Project mac progress report 11, MIT, 1974.

[Zus90] H. Zuse. *Software Complexity: Measures and Methods*. DeGruyter, Nueva York, 1990.