Análisis y Diseño

Daniel Rodríguez (daniel.rodriguezg@uah.es)

Índice

1. Introducción

En la fase de diseño, tomando como punto de partida los requisitos (funcionales y no funcionales), se pretende obtener una descripción de la *mejor* solución software/hardware que de soporte a dichos requisitos, teniendo no solamente en cuenta aspectos técnicos, sino también aspectos de calidad, coste y plazos de desarrollo. En teoría, se deberían plantear varios diseños alternativos que cumplan con los requisitos, y la elección final debería tomarse también de acuerdo a criterios de coste o esfuerzo de desarrollo, y criterios de calidad intrínsecos como por ejemplo la facilidad de mantenimiento. Es por tanto que en esta fase se pasa del *qué* obtenido en la fase de requisitos, al *cómo*, lo cual es el objetivo de la fase de diseño. Un ejemplo muy simple en un sistema de facturación, el qué, un requisito, podría ser que las facturas estuviesen ordenadas por código postal para que la empresa de correos haga una rebaja. En el cómo, se encontrarían el diseño del modulo de ordenación y la comunicación con la facturación, y más bajo nivel, la decisión del tipo de algoritmo de ordenación entre los que cumplan los requisitos (funcionales y no funcionales). Además con la fase de diseño, se entra en un a un ámbito más técnico ya que los productos del diseño son documentos y artefactos orientados a ingenieros del Software, y no a la comunicación con los clientes o usuarios, aunque puede darse el caso de necesitar generar documentación de diseño para los clientes. El estándar IEEE 610.12-1990 define diseño como:

Definición [18]
El diseño del software tiene las dos siguientes acepciones: ▪ El proceso para definir la arquitectura, los componentes, los interfaces, y otras características de un sistema o un componente. ▪ El resultado de este proceso.

Esta definición concreta algunos de los elementos importantes en los

diseños software como es la arquitectura, es decir, es la descomposición de un sistema en sus componentes e interfaces. Estos conceptos se tratan en más profundidad en las siguientes secciones. Además, en lo referente a la estructura del propio proceso de diseño, este se descompone en dos subprocesos:

- Diseño de la arquitectura o de alto nivel, en el cual se describe como descomponer un sistema y organizarlo en los diferentes componentes (la arquitectura del software)
- Diseño detallado, el cual describe el comportamiento específico de dichos componentes de software.

2. Conceptos fundamentales de diseño

En esta sección se cubren los conceptos fundamentales, tales como abstracción y modularidad, propiedades y reglas se tiene que cumplir para que los módulos se consideren modulares (descomposición de un sistema) y faciliten los criterios de calidad.

2.1. Abstracción

Como en el resto de los problemas de ingeniería, en el desarrollo de una solución de software, la solución se representará de forma abstracta con diferentes grados de detalle. Desde un nivel de abstracción alto y refinando la misma hasta conseguir un nivel de detalle próximo a la implementación. Se pueden diferenciar tres tipos fundamentales de abstracciones en el desarrollo de un sistema:

- Abstracción de datos. Define un objeto que compuesto de un conjunto de datos. La abstracción *Cliente*, por ejemplo incluirá todos los datos tales como *nombre*, *dirección*, *teléfono*, etc.

- Abstracción de control. Define a un sistema de control de un software sin describir los datos internos que permiten operar al mismo. Por ejemplo una abstracción de control típica es el *semáforo* para describir la coordinación en el funcionamiento de un sistema operativo.

- Abstracción procedimental. Es aquella que se referiere a la secuencia de pasos que constituyen un proceso determinado, por ejemplo un algoritmo de ordenación.

2.2. Componentes e interfaces

El objetivo del diseño es la especificación de componentes, módulos o fragmentos software tales como clases, y cómo se comunican sin describir sus detalles internos. Estas comunicaciones se expresan mediante la especificación de las operaciones que los componentes exponen para que otros puedan usarlos (interfaces).

Definición
Un **componente** es una parte funcional de un sistema que oculta su implementación proveyendo su realización a través de un conjunto de interfaces.

Por tanto, un componente es generalmente un elemento reemplazable y autocontenido dentro de una arquitectura bien definida que se comunicará con otros componentes a través de interfaces.

Definición
Una **interfaz** describe la frontera de comunicación entre dos entidades software, es decir, de cómo un componente interactua con otros componentes.

Nótese que el término interfaz se utiliza también con otros significados que nada tienen que ver, como el caso de la interfaz de usuario.

2.3. Descomposición y modularización

La descomposición y modularidad son consecuencia de la complejidad de los problemas, y de la necesidad de hacer manejable el desarrollo de la solución de los mismos. Para abarcar el desarrollo del sistema complejo, el problema se divide en subproblemas más fácilmente manejables, que integrados formarán la solución al sistema completo. Más formalmente, Meyer [30] ha definido una serie de propiedades para evaluar la modularidad:

Descomposición Esta propiedad permite definir componentes de alto nivel en otros de bajo nivel. Esta descomposición puede ser recursiva, es decir, aplicando la máxima de divide y vencerás.

Composición La composición se puede ver como el problema inverso a la descomposición. Un modulo de programación preserva la composición modular si facilita el diseño de elementos de programación que se pueden ensamblar entre sí para desarrollar aplicaciones. Un ejemplo de composición modular son los componentes ya desarrollados (COTS, *Commercial-off-the-Shelf*), como se describen en la sección 4.2. La composición modular está directamente vinculada con la reutilización, mediante mecanismos que permitan diseñar elementos que respondan a funcionalidades bien definidas y reutilizables en diversos de contextos.

Comprensión Un método de programación preserva la comprensión modular si facilita el diseño de elementos de programación que se pueden interpretar fácilmente sin tener que conocer el resto de los módulos. Un aspecto fundamental de la comprensión es la documentación y en el caso particular de los componentes COTS, la gestión e indexación de los mismos para facilitar su reutilización.

Continuidad La continuidad se refiere a que un pequeño cambio en la especificación debe conllevar en la implementación un cambio igualmente pequeño. Una de las leyes de Lehman sobre la evolución del software (ver sección **??**), afirma que si un proyecto esta *vivo* inevitablemente evolucionará y cambiará. Por tanto, es importante que los cambios en

los requisitos repercutan en un número limitado y localizado de módulos.

Protección Esta propiedad consiste en que los efectos de las anomalías de ejecución queden confinados al módulo donde se produjo el error, o que se propaguen a un número limitado de módulos con los que interactúa directamente. Un ejemplo de protección se da en la validación de datos introducidos por parte del usuario, dicha validación debería hacerse en los módulos que tratan la entrada, sin permitir que los datos incorrectos se propaguen a aquellos módulos donde se procesan.

2.4. Medición de la modularidad

Las propiedades y reglas mencionadas anteriormente buscan que las dependencias entre módulos sean mínimas. La modularidad se suelen medir con los conceptos de acoplamiento y cohesión definidos a continuación.

2.4.1. Acoplamiento

El acoplamiento mide el grado de interconexión existente entre los módulos en los que se ha dividido el diseño de la arquitectura de un sistema software.

El objetivo es conseguir un acoplamiento bajo entre módulos, también llamado débil, genera sistemas más fáciles de entender, mantener y modificar, ya que cambios en la interfaz de un componente afectarían un número reducido de cambios en otros componentes. Por tanto, debemos acercarnos al mínimo número de relaciones posibles entre todos los módulos en la que un módulo sólo se comunicaría con otro módulo, es decir, $(n-1)$ comunicaciones entre módulos, donde n es el número de módulos de un sistema (figura 1(a)). Por el contrario debemos alejarnos del máximo máximo número de conxiones, $(n\dot{(}n-1)/2)$, donde se produce un acoplamiento fuerte (figura 1 (b)).

Además del número de conexiones, el grado de acoplamiento puede depender de la la complejidad de las conexiones, los lugares donde se realicen

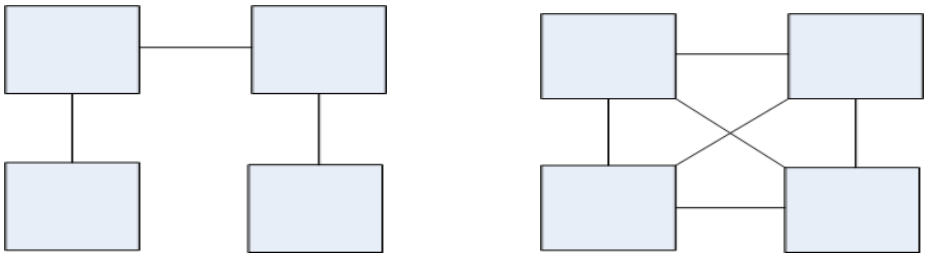


Figura 1: (a) Acoplamiento débil; (b) Acoplamiento fuerte

las referencias a los módulos, el volumen y tipo de datos que intercambian los módulos.

2.4.2. Cohesión

Otro aspecto fundamental del diseño, también derivado de una concepción modular del mismo, es la cohesión. Un subsistema o módulo tiene un alto grado de cohesión si mantiene unida funcionalidad común. Por ejemplo, un clase dedicada al manejo de fechas, tiene sólo operaciones relacionadas con las fechas y no otras funcionalidades que usen fechas como podrían ser la gestión de cumpleaños, etc.

Por tanto, el objetivo es diseñar módulos robustos y altamente cohesionados cuyos elementos estén fuerte relacionados entre si buscando la **cohesión funcional**, en la que un módulo realiza operaciones bien definidas y suscritas a una funcionalidad requerida facilitando las propiedades compresibilidad, reutilización.

Cuando los módulos agrupan funcionalidad por otros motivos que no sean funcionalidad la cohesión no será optima. Por ejemplo, *cohesión secuencial* en la que la funcionalidad se agrupa por la secuencia de ejecución, *cohesión por comunicación* al compartir los mismos datos, o la peor de todas, *cohesión por coincidencia* agrupando funcionalidad sin orden (cajón desastre).

Reglas de Meyer para la modularidad.

Meyer [30] propuso 5 reglas que hacen referencia a la conexión entre programas y a la interacción entre módulos, que se deben preservar para garantizar la modularidad.

- **Correspondencia directa**. Debe existir una relación coherente (correspondencia) entre el dominio del problema y la solución.
- **Limitación en el número de interfaces**. Se debe reducir al máximo el número de comunicaciones entre módulos.
- **Limitación del tamaño de las interfaces**. Además el número de comunicaciones entre los módulos de la regla anterior, se debe limitar al mínimo el tamaño de la información intercambiada entre módulos.
- **Facilidad de reconocimiento de las interfaces**. La comunicación entre módulos tiene que ser pública y reconocible, es decir, las interfaces deben ser explícitas facilitando así el diseño de la interfaz.
- **Ocultación de la información**. Se debe obtener una modularización que aísle los cambios al menor número posible de módulos. En la etapa de diseño de un módulo, se especifica el conjunto de propiedades del módulo que constituirán la información a la cual tendrán acceso los otros módulos. Dichas propiedades públicas compondrán la interfaz del módulo [31].

2.5. Arquitectura de sistemas

Con los criterios comentados anteriormente, podemos definir más formalmente el concepto de arquitectura y aunque es un concepto muy amplio, el estándar IEEE 1471-2000 (*IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*), también publicado como

estándar ISO/IEC 42010:2007, definen la arquitectura del software como un acuerdo de mínimos.

Definición (IEEE 1471-2000)
La organización fundamental de un sistema plasmado en sus componentes, las relaciones entre estos y con el entorno, y los principios guiando su diseño e implementación.

Éste es el primer estándar sobre arquitectura software y como su nombre indica, define las practicas recomendadas, que siguiendo la terminología IEEE, no es normativa, es decir, su adopción e interpretación son responsabilidad de las organizaciones que las implementan. Por tanto, el estándar sólo describe un marco general en el que aborda los siguientes puntos:

- Identificación a todas las personas interesadas en el proyecto y sus intereses.
- Selección e identificación de los puntos de vista para los distintos intereses.
- Documentación de las vistas de la arquitectura que satisfacen los puntos de vista
- Documentación las inconsistencias entre vistas
- Proporcionar la base para las decisiones arquitecturales.

Como se ha comentado anteriormente, debido a la complejidad de un sistema software en cuanto a la especificación, diseño y construcción, requiere diferentes perspectivas. El diseño arquitectural nos permite organizar diferentes puntos de vista y controlar el desarrollo de un sistema mediante organización del sistema, evaluación y selección tanto de los aspectos estructurales para cumplir con los requisitos funcionales, como de los aspectos no estructurales para cumplir con los requisitos no funcionales como por ejemplo rendimiento, capacidad de adaptación, reutilización, etc.

En el diseño de un sistema, se necesitan diferentes vistas para modelar las diferentes partes constituyentes de un sistema, descripción de los subsistemas, componentes y las interrelaciones entre ellos. Las diferentes aproximaciones al diseño pueden describir los sistemas con distintas técnicas, y dependiendo del tipo de proyecto, podríamos considerar:

- Arquitectura funcional, describiendo las distintas funciones del sistema. Es importante destacar que el criterio de división en subsistemas puede ser de dos tipos:
  - Criterios verticales o funcionales de usuario, en los cuales la descomposición agrupa funcionalidades (reflejadas en los requisitos funcionales). Por ejemplo, en una aplicación para la enseñanza virtual, podrían considerarse subsistemas la gestión de contenidos, gestión de comunicaciones con los alumnos, y por otro lado, el subsistema de administración de alumnos, notas, etc.
  - Criterios transversales a las funcionalidades. Estos incluyen todos los aspectos que no son específicos de ciertos grupos funcionales. Siguiendo el ejemplo anterior, el subsistema de seguridad con el control y derechos de acceso, o un subsistema de persistencia agrupando los servicios relacionados con el almacenamiento.
- Estilo arquitectónicos que implementan la arquitectura funcional, por ejemplo describiendo las capas que lo componen, componentes y como interactúan entre ellos.
- La arquitectura de la base de datos como se describe en más detalle en la sección 3.2.
- Arquitectura hardware y de red que componen el sistema, por ejemplo en un sistema de telefonía influye la elección del tipo de red como ATM y sistemas operativos para tiempo real.

Por lo tanto, la arquitectura en la ingeniería del software trata de definir la estructura interna del software, pero deteniéndose en el nivel de los compo-

Figura 2: Estilo arquitectónico de procesamiento por lotes

nentes e interfaces que afectan a la interrelación que se está especificando, sus protocolos de comunicación y sincronización y distribución física del sistema, pero no trata, el diseño detallado o algoritmos.

2.5.1. Estilos arquitectónicos

Existen distintas formas de unir los distintos componentes de las arquitecturas de sistemas software, dependiendo de cómo son divididos los distintos módulos y como estos se comunican. Los estilos arquitectónicos se suelen describir como patrones con sus ventajas y desventajas. A continuación se clasifican los estilos arquitectónicos más comunes:

- Filtro-tubería (*Pipe-Filter*) o procesamiento por lotes. Se basan en flujos de datos, donde un componente transforma una entrada en una salida que a su vez es la entrada para otro componente (figura 2). Esta arquitectura es típica de *mainframes*, donde por ejemplo, en un sistema de facturación en COBOL, distintos módulos van transformando ficheros hasta conseguir las facturas finales.

- Orientación a Objetos. En este estilo, la arquitectura se representa principalmente mediante el diagrama de clases, que se podría definir como un grafo cuyos nodos son objetos y los arcos son los conectores que comunican los objetos mediante los métodos. Este estilo se ve en detalle más adelante.

- Arquitectura basada en eventos (Invocación implícita) . Los componentes anuncian sus eventos y otros registran su interés en ser notificados cuando ocurren ciertos eventos causando la invocación de procedimientos en otros componentes. Por ejemplo, en las interfaces de

Figura 3: Arquitectura en capas de un sistema operativo

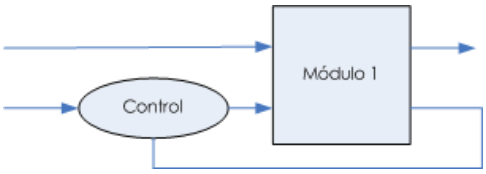


Figura 4: Ejemplo de arquitectura de control

usuario, eventos como el pulsar el ratón, disparará la ejecución de ciertas funciones

- Arquitecturas basadas en capas. Los componentes del sistema están organizados jerárquicamente por capas, como si fuera un cebolla donde las capas superiores acceden a los servicios de las capas inferiores. Por ejemplo, el diseño de un sistema operativo (figura 3).

- Sistemas basados en repositorios. En este estilo, existe una estructura central de datos y componentes independientes acceden a dicha estructura central de datos.

- Control de Procesos. En los procesos de control como mantenimiento de temperaturas o niveles de líquidos, se suele utilizar el estilo donde el bucle de control (*feedback loop*) podría ir con la entrada o después del proceso (figura 2.5.1).

- Procesos distribuidos. La distribución de la funcionalidad se divide en diferentes procesos generalmente distribuidos en diferentes máquinas. A su vez se pueden diferenciar en tareas *cliente-servidor*, por ejemplo la Web, y arquitecturas *par a par* (P2P – *Peer-to-Peer*) por ejemplo con sistemas de distribución de ficheros, TV o voz sobre IP (VoIP – *Voice over IP*)(figura 2.5.1).

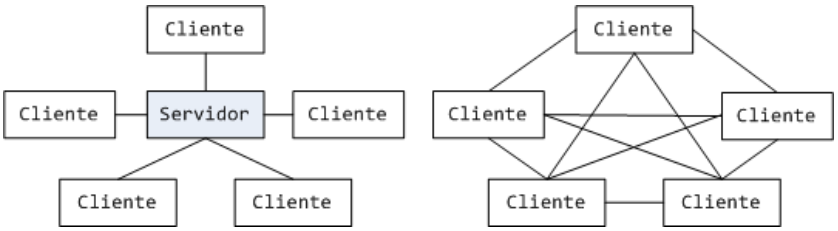


Figura 5: (a) Cliente-servidor; (b) *Peer-to-peer*

2.5.2. Lenguajes arquitectónicos

Los lenguajes arquitectónicos (*Aquitecture Description Language – ADL*) describen arquitecturas software mediante sus componentes, conectores y enlaces (comunicación) entre componentes. Cuando una arquitectura se describe mediante estos lenguajes, es posible tener herramientas para su verificación y prototipado rápido. Existen lenguajes arquitectónico de propósito general y otros de dominio específico (DSSA – *Domain-Specific Software Architectures*). Un ejemplo de lenguaje arquitectural es ACME (*Architecture Based Languages and Environments*)[1], aunque también puede considerarse UML (de mucho más extendido uso hoy día) para representar arquitecturas.

2.6. Notaciones de diseño

Existen multitud de notaciones de diseño utilizadas para describe los multiples artefactos necesarios para describir la arquitectura del sistema y su diseño. Muchas técnicas son exclusivas de los métodos estructurados, otras de los métodos orientados a objetos, pero es bastante habitual que un mismo proyecto utilize distintas notaciones para distintas partes del proyecto. Por ejemplo, hoy día lo más normal es basarse en la orientación a objetos y utilizar UML como lenguaje de modelado, pero parte de ese mismo proyecto podría modelar la base de data mediante Entidad/Relación, diagramas de flujo o tablas de decisión para describir algoritmos concretos, y *storyboards* para describir la navegación en las interfaces de usuario. En la siguiente sec-

[1] `http://www.cs.cmu.edu/~acme/`

ción 3 se describen las notaciones más relevantes junto con los métodos de diseño en sistemas estructurados, orientados a objetos o de bases de datos.

3. Métodos de diseño

En esta sección se proporciona una visión de cómo los métodos y técnicas de diseño han ido evolucionando con los métodos estructurados, la representación de los datos y la orientación a objetos.

3.1. Métodos estructurados

Con la aparición de los primeros lenguajes de programación fue posible desarrollar una gran cantidad de código, pero muy difícil de mantener, lo cual se denominó *código spaghetti*. Sobre finales de los años 1960, Dijkstra [9, 7] creó el paradigma la programación estructurada demostrando que todo programa puede escribirse utilizando únicamente (i) bloques secuenciales de instrucciones (ii) instrucciones condicionales y (iii) bucles. Con los lenguajes estructurados, aparecieron numerosos métodos de diseño y análisis estructurados como forma sistemática de desarrollo del software. La figura 6 muestra la evolución del análisis y diseño estructurados enumerando los métodos mas importantes.

Los métodos estructurados se basan en una aproximación descendente (*top-down*) que aboga por descomponer en niveles funcionales con sus entradas y salidas, desde una visión global hasta el nivel de detalle necesario para su implementación. Las característica más importantes de los métodos estructurados son la descomposición funcional, el modelado de los datos y la representación del flujo de información, que conforman las distintas vistas del sistema (figura 7 adaptada de [32]): la especificación de datos, la especificacion del proceso y la especificación del control. Las técnicas más comunes para describir estas vistas entre los métodos estructurados y que han perdurado en el tiempo se incluyen:

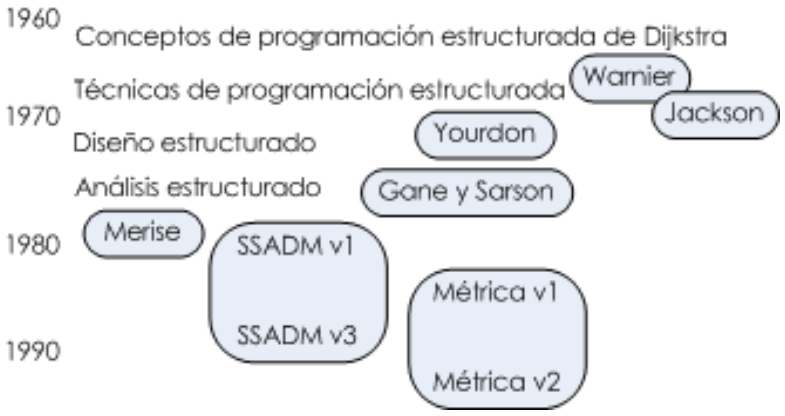


Figura 6: Evolución de los métodos estructurados

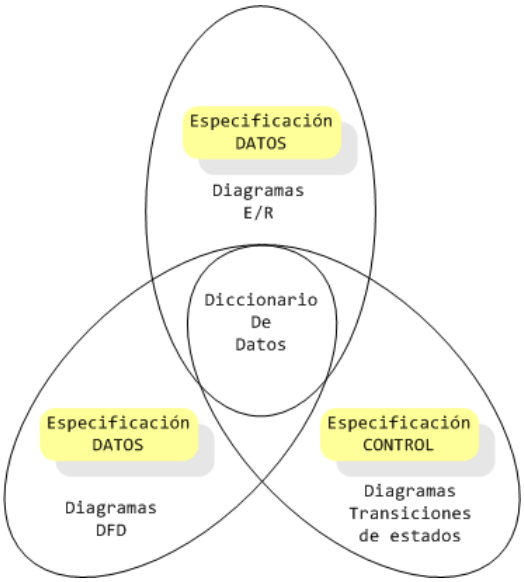


Figura 7: Vistas de los métodos estructurados

Diagramas de flujo de datos (DFD). Los DFDs (*Data Flow Diagrams* en sus siglas en inglés) se asemejan a un grafo que representa los flujos de datos y las transformaciones que se aplican sobre ellos. Los nodos representan procesos y los vértices las entradas y salidas a los mismos. Las entradas y salidas pueden ser externas al sistema y puede haber almacenes de datos entre los nodos. Los DFDs pueden descomponerse en otros sub-diagramas hasta llegar a un nivel de granularidad apropiado para el diseño siguiendo una aproximación descendente. La figura 8 muestra un ejemplo de descomposición de DFDs en posibles niveles. El nivel superior se denomina *nivel de contexto*, a los procesos que no se descomponen se les denomina *procesos primitivos*. Finalmente, para representar los DFDs existen diferentes notaciones aunque todas muy similares, los componentes de los DFDs son:

- *procesos* que describen las funcionalidades del sistema
- *almacenes* que representan los datos utilizados por el sistema
- *entidades externas* que representan la la fuente y/o destino de la información del sistema
- *flujos de datos* representando el movimiento de datos entre las funciones.

Diagramas Entidad-Relacion (E/R). Las relaciones entre los datos se suelen primero modelar mediante el modelo E/R que debido a su importancia, y a que es el principal diagrama de muchas aplicaciones orientadas a datos, se trata en más detalle en la siguiente sección.

Diccionarios de Datos (DD). Los DD contienen los datos utilizados en el sistema, de tal forma que todos los participantes del proyecto tengan la misma visión de la información manejada por el sistema. La figura 9 muestra un ejemplo de diccionario de datos. Los DD representan la información de los flujos de datos y los almacenes del sistema.

Diagramas de estructura Los diagramas de estructura (*Structure Chart*) son utilizados para representar gráficamente la estructura modular en

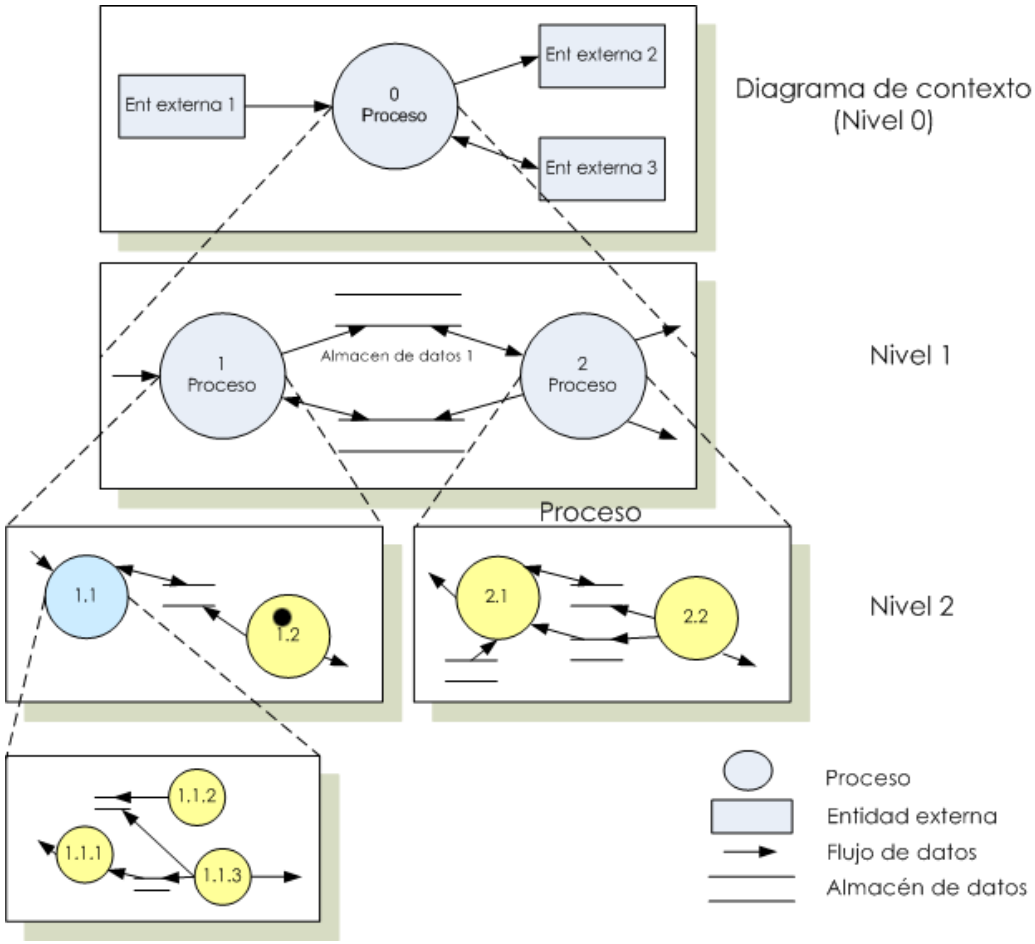


Figura 8: Niveles en los diagramas de flujo de datos

```
CLIENTE = @DNI + NOM_CLIENTE + DIRECCION + {NUM_TLF}
DIRECCION = [CALLE + NUM + PROV | APT_CORREOS ]
...
```

Notación:

`@`	Identificador
`= ...+ ...`	Composición
`{ }`	Iteración
`[...|...]`	Selección

Figura 9: Ejemplo de diccionario de datos

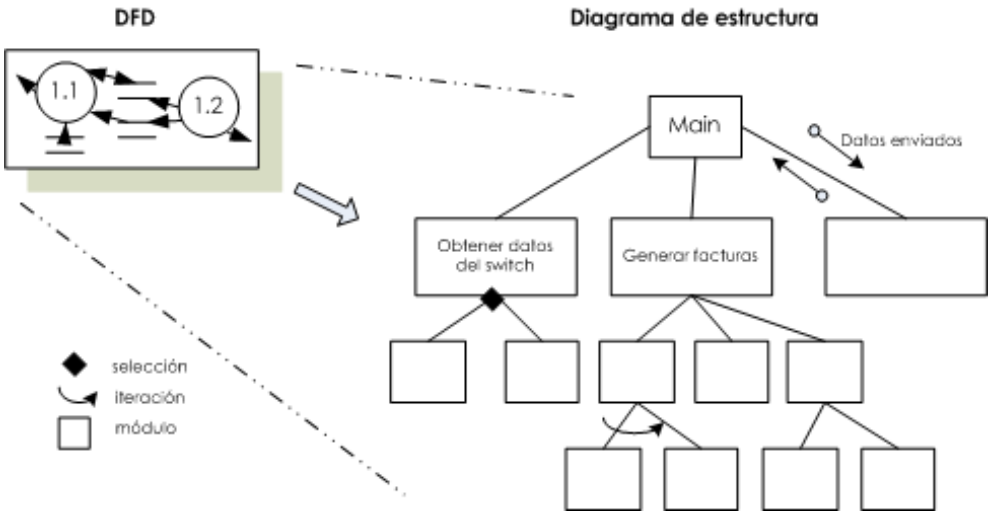


Figura 10: Transformación de DFDs a diagramas de estructura

un sistema estructurado, es decir, la jerarquía de módulos junto con las llamadas entre los módulos (control). Tal y como muestra la figura 10, los diagramas de estructura se obtienen a partir de los DFDs, transformándolos en árboles que descomponen el sistema en sus funcionalidades básicas. Además de la descomposición del sistema en sus módulos, los diagramas de estructura muestran información sobre la secuencia de ejecución (secuencial, repetitiva y alternativa), control y datos enviados o recibidos.

Finalmente, la descripción los de módulos individuales puede hacerse mediante numerosas técnicas, que incluyen:

- Las *tablas de transiciones*, también llamadas tablas de decisión definen en forma matricial reglas con acciones a realizar dadas ciertas condiciones como muestra el ejemplo de la figura 11.
- *Árboles de decisión*, similares a las tablas de decision pero en forma de árbol.
- *Diagramas de estados* son una técnica originalmente desarrolladas por D. Harel [15] para representar autómatas finitos, aunque son ampliamente utilizados en otras áreas como compiladores, sis-

	Regla		
	1	2	3
Condiciones			
Edad > 60	Sí	No	Sí
Estadio I	No	Sí	No
Estadio II	No	No	Sí
Acción			
Tratamiento A	X	X	–
Tratamiento B	–	X	X
Tratamiento C	–	–	X

Figura 11: Ejemplo de tabla de transición

temas de control, etc. También han sido adoptados en UML, y se describen en más detalle sección 3.3.

- *Pseudocódigo*, *diagramas de flujo*, etc.

Otros Métodos de diseño clásicos

Dentro de los métodos clásicos, también existen otras técnicas que por su importancia que tuvieron en el pasado se enumeran y describen brevemente:

Lenguajes de diseño de programas También conocidos por sus siglas en inglés, PDLs (*Program design languages*) expresan la lógica de los programas de manera similar al pseudocódigo y sus interfaces. Surgieron como sustitutos de los diagramas de flujo y tenían la ventaja de poder ser validados mediante herramientas [4].

Métodos estructurados orientados a datos Muchos de los programas en lenguajes procedurales como COBOL, estaban orientados a la transformación y manipulación de datos en registros, siendo los datos en núcleo del sistema. Métodos representativos de este grupo fueron Warnier [38] y JDM (Jackson Development Method). El método JDM [19] fue relevante al ser la base de otros métodos posteriores, inicialmente basado en 3 las estructuras básicas para modelar programas del teorema demostrado por Dijikstra (secuenciación, iteración y selección), el método fue después extendido a otras fases del ciclo de vida y automatización en la generación de código.

JAD (*Joint Application Development*) Esta técnica fue muy popular por lo que se extendió a la fase de diseño, sobre como abordar las reuniones en la fase de diseño en una misma localización o incluso reuniones virtuales.

3.2. Métodos orientados a datos

Un gran porcentaje de aplicaciones y prácticamente las aplicaciones de gestión gestión manejan una serie de datos bien en ficheros o bien mediante

bases de datos. El modelo Entidad/Relación (E/R) no sólo es una de las formas más comunes de expresar los resultados del análisis de un sistema en etapas tempranas del desarrollo, sino que además suele ser el punto de partida para el diseño de las bases de datos en los sistemas.

Una vez obtenido diagrama E/R (modelo conceptual), este se normaliza para obtener el diseño lógico de la base de datos. Ello se consigue aplicando un conjunto de reglas a cada uno de los elementos del modelo conceptual para que, conservando la semántica, se transforme en un elemento del modelo lógico, generalmente el modelo relacional. En muchos casos la transformación es directa porque el concepto es el mismo (por ejemplo, las entidades pasan a ser tablas en la base de datos), pero otras veces no existe esta correspondencia, por lo que es necesario realizar un proceso de transformación. Por ejemplo, la figura 12 muestra simplificadamente la transformación desde un modelo conceptual en E/R a tablas en una base de datos; por ejemplo, en este caso la relación*participa* $N : M$ – muchos a muchos – se transforma en una tabla intermedia, conservando así la semántica de la relación. Finalmente, el modelo físico refleja aspectos de implementación tales como índices, tipos de estructuras, etc.

3.3. Diseño orientados a objetos

El paradigma de la orientación a objetos (OO) se remonta a 1967, año en que dos físicos noruegos O.-J. Dahl y K. Nygaard desarrollaron los conceptos básicos de este tipo de programación en un lenguaje llamado SIMULA donde vieron la necesidad de unir datos y funciones para ejecutar multiples simulaciones. Entre los lenguajes influenciados por SIMULA se encuentran Smalltalk, C++, Eiffel, Java y C#. Hoy en día, es es el paradigma de programación más utilizado y los lenguajes orientados a objetos cumplen las propiedades fundamentales de la orientados a objetos (ver tabla adjunta) permitiendo mejorar la calidad del software producido.

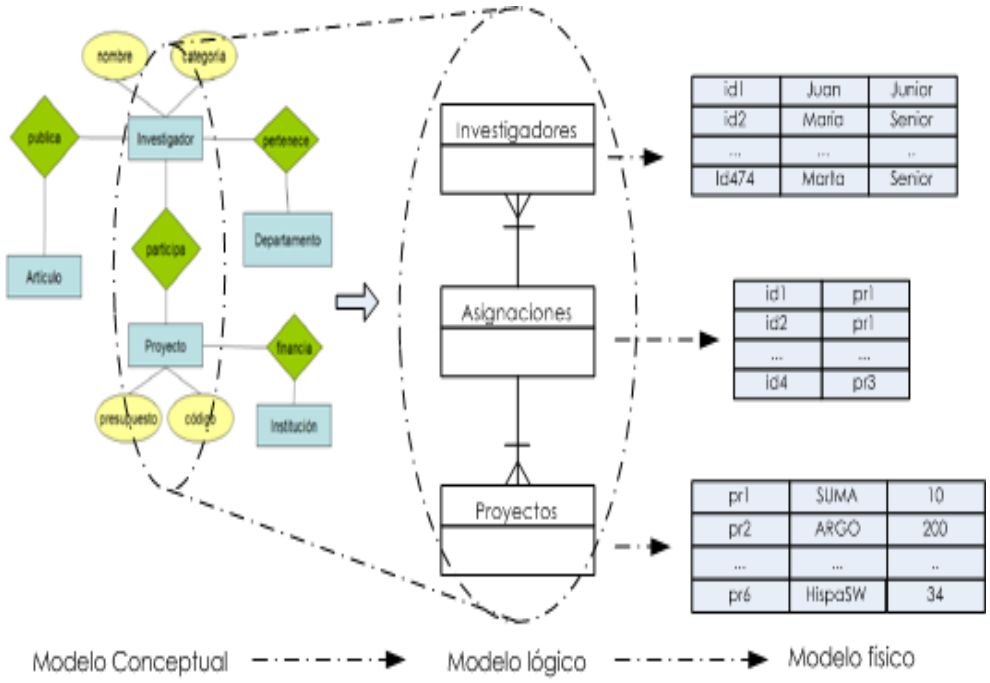


Figura 12: Transición desde el modelo E/R al diseño físico de bases de datos

Propiedades de la orientación a objetos

Abstracción Se reduce la complejidad del dominio abstrayendo hasta el nivel adecuado. En la orientación a objetos la abstracción se representa mediante el concepto de clase, que representa un conjunto de objetos concretos, llamados instancias con sus propiedades y operaciones.

Herencia Esta propiedad permite definir una clase a partir de una o más clases tal que la subclase tenga hereda las características de la(s) superclase(s), más las características propias de la subclase.

Encapsulamiento Los datos y operaciones de una clase están agrupados tal que los clientes de una clase sólo necesitan conocer la interfaz pública de la misma, es decir, los prototipos de las operaciones pero no cómo están implementadas.

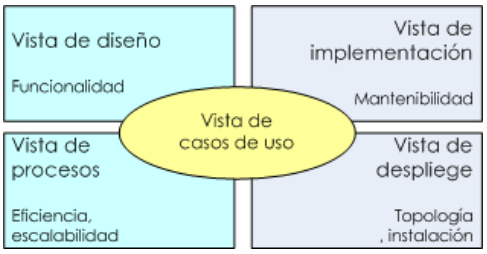
Polimorfismo Es la propiedad por la que un mismo nombre de método esta asociado a distintos comportamientos, pudiendo ser estático o dinámico. El *polimorfismo estático* consiste en que operaciones con el mismo nombre pueden realizarse sobre distintos tipos de parámetros. El *polimorfismo dinámico*, está relacionado con la herencia y se llama asignación tardía o dinámica (*dynamic binding*) que resuelve cuál es el método a llamar durante la ejecución dependiendo del tipo de la referencia.

Al igual que con los métodos estructurados, también con la aparición de los lenguajes orientados a objetos surgieron numerosos métodos de diseño orientados a objetos, con distintas notaciones y herramientas, dando lugar a lo que se conoció como la guerra de las metodologías. Entre las metodologías, destacaron Booch, OMT (*Object Modeling Technique*) y OOSE/Objectory cuyos principales autores, G. Booch, J. Rumbaugh e I. Jacobson respectivamente, se unieron dando lugar al lenguaje unificado de modelado (UML – *Unified Modeling Language*) y el *Proceso Unificado* (ver sección **??** y cuadros asociados). UML ha aglutinado las más importantes notaciones de distintas metodologías convirtiéndose en la notación estándar y de referencia para el diseño orientado a objetos. Además, como afirman sus autores, UML es un lenguaje para visualizar, especificar, construir y documentar sistemas en general pero particularmente adaptado a sistemas software construidos mediante el paradigma de la orientación a objetos.

Las 4+1 vistas de Kruchten [23]

En procesos como UP y OpenUP, la arquitectura de un sistema se describe mediante 4 vistas complementarias más la vista de los casos de uso complementado la información de las otras vistas:

- La *vista de diseño* muestra como el sistema lleva a cabo los requisitos funcionales mediante la descomposición del sistema en sus elementos como clases y su comunicación entre estos.
- La *vista de implementación* describe la organización del sistema en módulos, componentes y paquetes cubriendo el ensamblado del sistema y la gestión de configuración, reuso y portabilidad.
- La *vista de procesos* describe los procesos y cómo estos se comunican, y aún menor nivel de detalle, los hilos de control en las clases (concurrencia y distribución de procesos).
- La *vista de despliegue* se utiliza para representar un conjunto de nodos físicos formando la topología hardware del sistema.
- La *vista de los de casos de uso* describe los requisitos funcionales del sistema, utilizados para complementar las otras vistas y en todo el ciclo de vida, por ejemplo, para crear los casos de prueba.



Aunque UML es independiente del proceso que se siga, está generalmente ligado a procesos iterativos e incrementales como el Proceso Unificado u Open UP, descritos en el capitulo **??**. En estos procesos y otros, los sistemas orientados a objetos se describen utilizando las 4+1 vistas (ver cuadro) de

Kruchten [23] mediante *modelos* y *diagramas*. Un **modelo** captura una vista de un sistema abstrayendo y describiéndolo un apropiado nivel de detalle. Los modelos a su vez se representan mediante **diagramas**, que son es representaciones gráficas de un conjunto de elementos de modelado y sus relaciones.

En UML, los diagramas están clasificados en dos grupos:

- Los *Diagramas de estructura* reflejan la estructura física (estática) del sistema por medio de sus clases, métodos, atributos, interfaces, paquetes, etc. y sus relaciones.
- Los *Diagramas de comportamiento* reflejan la forma en los distintos elementos del sistema interactúan, colaboran, cambian de estado durante la ejecución del sistema para proveer la funcionalidad requerida.

En UML versión 2 existen 13 tipos de diagramas que formarían los distintos modelos para cada una de las vistas. A continuación se proporciona una visión general de los diagramas más importantes y un resumen en la tabla 1. Además a continuación se describen de manera muy general los diagramas más relevantes de UML sin entrar en detalle, ya que queda fuera del alcance de este libro el describir toda la notación exhaustivamente. Además, es importante resaltar que no es necesario utilizar todos los tipos de diagramas en los proyectos, al igual que generalmente no se especifica completamente los diagramas al 100 %, es decir, hay un balance entre completitud y eficiencia desde el punto de vista del diseñador/a.

Diagrama de casos de uso Los diagramas de Casos de Uso (UC) son una técnica para la captura y especificación de requisitos, principalmente requisitos funcionales, que un sistema proporciona en un entorno. Aunque estrictamente hablando la técnica de los UC no pertenece al enfoque orientado a objetos, ha sido adoptada por UML y por ende, del diseño orientado a objetos. Una vez especificados los requisitos, las distintas vistas de un sistema, por ejemplo usando las 4+1 vistas de Kruchten, se pueden describir con el resto de diagramas de UML.

Cuadro 1: Diagramas de UML 2

Diagramas de estructura	
Clases	Es el diagrama más importante que muestra un conjunto de clases, interfaces y colaboraciones con sus relaciones.
Objetos	Muestra un conjunto de objetos y sus relaciones en un estado concreto. Generalmente, representan la instanciación de un diagrama de clases en un determinado punto en el tiempo.
Componentes	Describe los componentes que componen una aplicación con sus interrelaciones e interfaces publicas.
Estructura compuesta	Permite visualizar de manera gráfica las partes que definen la estructura interna de un clasificador, incluyendo sus puntos de interacción con otras partes del sistema.
Paquetes	Muestran la organización en paquetes de los diferentes elementos que conforman el sistema, de forma que se puede especificar de manera visual el nombre de los espacios de nombres.
Despliegue	Muestra la arquitectura física de un sistema, nodos en sus entornos de ejecución y como se conectan.
Diagramas dinámicos	
Actividad	Muestra los procesos de negocio y flujos de datos
Comunicación	Muestra la organización estructural de los objetos y el paso de mensajes entre ellos.
Interacción	Variante del diagrama de actividades para mostrar el flujo de control de un sistema o proceso de negocio.
Secuencia	Modela la secuencia lógica de mensajes entre participantes (generalmente clases) temporalmente.
Máquinas de estados	Describe los distintos posibles estados de un objetos junto con sus transiciones (generalmente utilizado para representar el comportamiento de clases complejas.
Tiempos	Muestra los cambios de estado o condición de objetos a eventos externos sobre el tiempo.
Casos de Uso	Muestra los casos de uso, actores y sus relaciones.

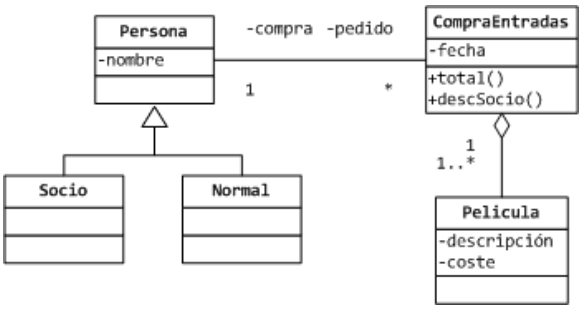


Figura 13: Diagrama de clases

Diagrama de clases El diagrama de clases es el modelo estático más importante. Muestra las clases, interfaces y sus relaciones. Las *clases* abstraen las características comunes a un conjunto de objetos, por ejemplo la clase `Persona` podría abstraerse con una serie de propiedades como nombre, fecha de nacimiento, etc. Los objetos son instancias concretas de las clases, por ejemplo 'Joe Doe' sería un objeto de la clase `Persona`. Las objetos o entidades siempre necesitan colaboran entre ellas para realizar la funcionalidad requerida, las relaciones como conexiones entre los elementos se denominan *asociaciones*. Hay diferentes tres tipos de asociaciones: herencia, agregación y composición.

Las clases se representan mediante un rectángulo dividido en 3 secciones, la primera tiene el *nombre* de la clase, la segunda sus *atributos* (propiedades) y la tercera las *operaciones* (métodos) de la clase indicando los servicios que proporciona la clase. La figura 13 muestra un ejemplo de diagrama de clases, por ejemplo, la clase `CompraEntradas` almacena las ordenes de compra de entradas de un cine, la `fecha` sería el atributo y `total()` y `descSocio()` los métodos de la clase. La clase tiene como asociaciones `Persona` y `Pelicula` como asociaciones.

A una instanciación de un diagrama de clases mostrando los objetos y sus relaciones en un instante concreto, i.e., un determinado punto en el tiempo, se le denomina **diagrama de objetos**.

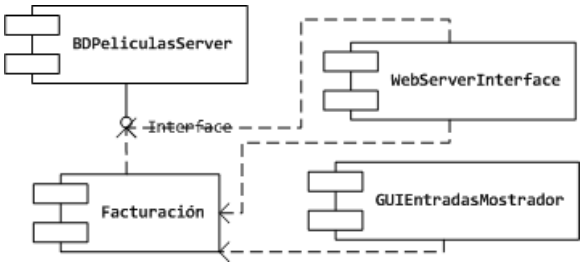


Figura 14: Diagrama de componentes

Diagrama de componentes Estos diagramas muestran los elementos físicos del sistema tales como librerías, APIs, archivos, etc. y sus relaciones. Un componente se pueden agrupar, por ejemplo, varias clases e interfaces representado cierta funcionalidad agrupada en una librería dinámica o un programa ejecutable.

La figura 14 se muestra un ejemplo de diagrama de componentes. La asociación (mostrada mediante una flecha discontinua muestra la dependencia de un paquete sobre otro y el círculo representa una interfaz.

Diagramas de interacción Estos diagramas muestran como interactuan (mediante el paso de mensajes) los objetos o clases entre si y pueden ser de dos tipos:

- **Diagramas de secuencia**. Resaltan el paso de mensajes en el tiempo, es decir, su ordenación temporal.
- **Diagramas de comunicación** (antes llamados diagramas de colaboración). Son equivalentes a los diagramas de secuencia, pero destacan la organización en la comunicación entre los objetos, es decir, como los mensajes unen los objetos en lugar del tiempo.

La figura 15 muestra un ejemplo muy simplificado de diagrama de secuencia (a) y su diagrama de comunicación equivalente (b) resaltado el paso de mensajes en el tiempo y o la secuencia de mensajes respectivamente.

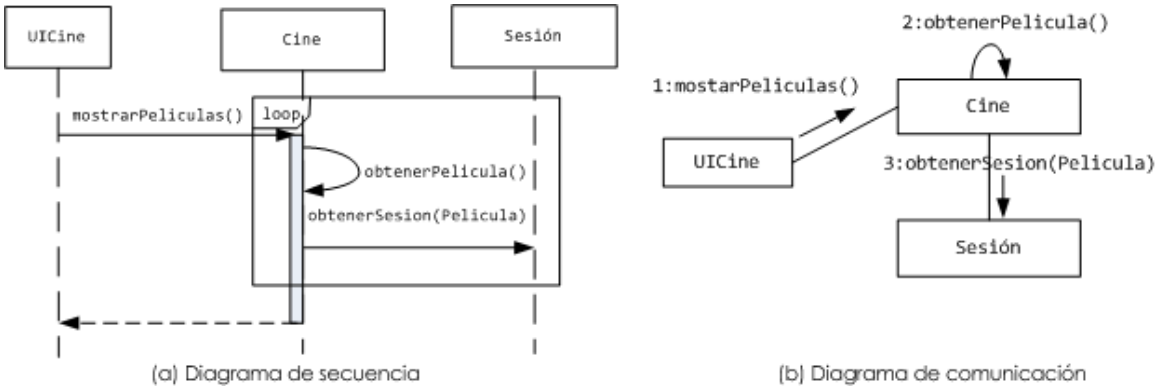


Figura 15: Ejemplos de diagrama de secuencia y de comunicación

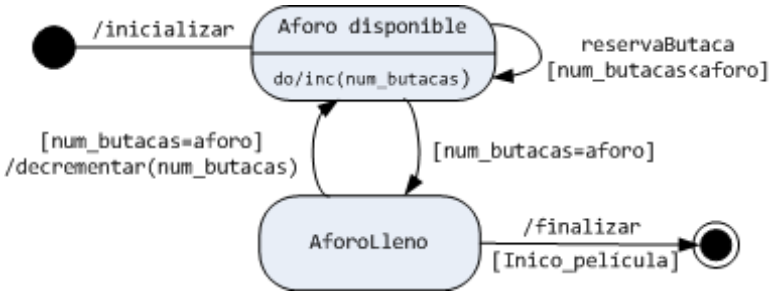


Figura 16: Diagrama de estados

Diagramas de estados Los diagramas de estados, originalmente creados por Harel [15], representan autómatas de estados finitos mostrando los estados por los cuales puede pasar un objeto, junto con los cambios que permiten pasar de un estado a otro. Son útiles para representar objetos complejos con multiples estados, por ejemplo controladores. La figura 16 muestra un ejemplo de diagrama de estados donde los nodos representan estados de la clase y los arcos representan eventos que realizan la transición entre los estados; la barra debajo de los eventos indica acciones y el texto entre corchetes representa las condiciones o guardias para realizar la transición entre estados.

Diagrama de actividad Un diagrama de actividad muestra paso a paso la estructura de un proceso o algoritmo en forma de flujo de control. Es decir, muestra cómo y qué hacen las acciones, así como su secuencia, pudiendo además mostrar actividades paralelas, tomas de decisión, etc.

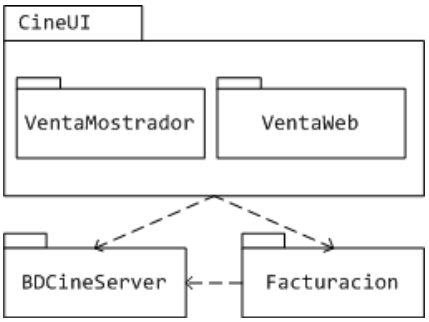


Figura 17: Ejemplo de diagrama de paquetes

Aunque la notación es similar a los diagramas de estados, los nodos de los diagramas de actividad muestran actividades, mientras que en los diagramas de estados los nodos muestran estados "fijos". Además, no son específicos de la orientación a objetos ya que suelen utilizarse para especificar procesos de negocio y con notaciones para representar flujo de control y paralelismo.

Diagrama de paquetes Muestra la organización lógica en paquetes de los diferentes elementos que conforman el sistema, de forma que se puede especificar de manera visual el nombre de los espacios de nombres. La figura 17 muestra un ejemplo de diagrama de paquetes.

Diagrama de despliegue Muestran la arquitectura del hardware del sistema y la distribución física de los componentes software en el hardware. La figura 18 muestra un ejemplo de diagrama de despliege.

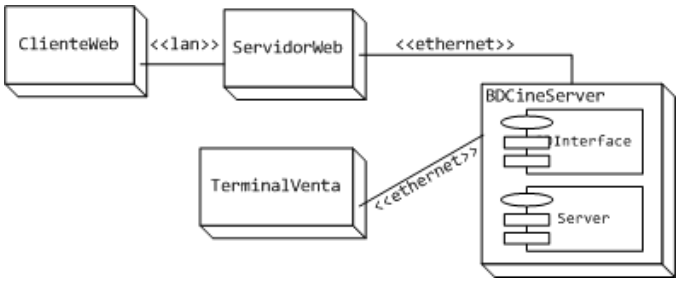


Figura 18: Ejemplo de diagrama de despliege

Resumen del diseño orientado a objetos Se ha proporcionado una vision general, sin entrar en profundidad el las notaciones, o en cómo unir los componentes, clases, etc. de UML y su aplicación depende mucho de la experiencia. Se proporcionan, sin embargo, unos principios básicos en el cuadro asociado.

Principios del diseño orientado a objetos

Se han definido unos principios que permiten alcanzar los objetivos de una bajo acoplamiento y una alta cohesión. Estos principios son:

- **Principio abierto-cerrado** [30]. Un módulo debería ser a la vez abierto y cerrado: abierto para poder ser extendido y cerrado para ser modificado. En otras palabras, un módulo debe poder extender su funcionalidad sin necesidad de modificar su código fuente por medio de interfaces bien definidas.
- **Principio de responsabilidad única**. Introducido DeMarco [8] en el diseño estructurado, esta relacionado con el concepto de cohesión en el sentido de que una responsabilidad es una razón para el cambio y por tanto, cada clase debe tener una única responsabilidad.
- **Principio de separación de la interfaz** Los clientes no deberían ser forzados a depender de interfaces que no utilizan. En otras palabras, es preferible tener muchas interfaces específicas que una sola interfaz de propósito general.
- **Principio de sustitución de Liskov** [25]. La herencia ha de garantizar que cualquier propiedad que sea cierta para los objetos de la clase, también lo son para los objetos de sus subclases. En otras palabras, las subclases, deben de poder sustituirse por la clase base.
- **Ley de Demeter**. Descrita por Lieberherr y Holland [26], esta ley busca mejorar el acoplamiento entre clases y establece que cada unidad debería tener solamente un conocimiento limitado sobre otras unidades, sólo las relacionadas a la unidad. Informalmente descrita como *"no hables con extraños"*.
- **Principio de inversión de dependencias**. Descrito por Martin [27] establece que (1) los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de las abstracciones. (2) Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.
- **Principio de dependencias estables**. [28] Las dependencias entre paquetes en un diseño deberían ir encaminadas a la estabilidad de los paquetes. La estabilidad se mide por el número de dependencias de entrada y salida de un paquete. Cuantas más dependencias de entrada tiene un paquete, más estable necesita ser, ya que cualquier cambio afectará a todos los paquetes que dependen de él. Otra regla es importante es evitar ciclos en la estructura de paquetes.

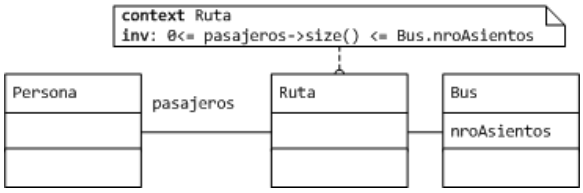
Finalmente, el paradigma de la orientación a objetos sigue evolucionando con técnicas como la *programación orientada a aspectos*, al igual que los estándares, notaciones y herramientas con ejemplos como el lenguaje de restricción de objetos (OCL) y la ingeniería dirigida por modelos (MDE). Ambos se describen brevemente en los cuadros asociados.

El Lenguaje de Restricción de Objetos (OCL) – [36]

UML incorpora OCL (*Object Constrain Language*) para extender la semántica de los elementos de UML con un lenguaje de lógica primer orden pensado para:

- Especificar *invariantes* en las clases tipos o interfaces
- Describir *precondiciones* y *postcondiciones* en los métodos, guardas de los diagramas de estados, actividad, etc.
- Lenguaje de navegación y reglas de formación.
- Definir restricciones de operaciones.

El siguiente ejemplo muestra un diagrama de clases y una restricción de cardinalidad en OCL que no puede mostrarse utilizando UML, donde el número de pasajeros está limitado por el número de asientos disponibles en el autobús.



Aparte del diseño, OCL se ha incorporado en herramientas capaces de traducir restricciones OCL en aserciones del lenguaje de programación demostrando su utilidad en la verificación y pruebas de programas.

Arquitectura dirigida por modelos (MDA)
La MDA (*Model Driven Architecture*) es un *framework* ideado para desarrollo mediante modelos. Basado en UML, el desarrollo se realiza en tres niveles: • Primero, se define un modelo con un alto grado de abstracción, independiente de la tecnología sobre la que se va a desarrollar, lo que se llama PIM (*Platform Independent Model*). • Segundo, el modelo PIM se transforma en modelos específicos de la plataforma, llamados PSM (*Platform Specific Model*). • Finalmente, se transforman los PSM en código ejecutable. La transformación entre los niveles de abstracción, de PIM a PSM y de PSM a código, se realiza mediante herramientas, automatizando el proceso de transformación. Por lo que el desarrollo se limita al modelado de alto nivel mediante UML y estándares relacionados como OCL. Aunque se empiezan a ver algunas herramientas que lo hacen realidad, el principal inconveniente es el esfuerzo necesario en especificar los modelos en el nivel de detalle necesario que permita su çompilaciónz la dificultad de proponer modelos independientes y la transformación entre modelos.

4. Otras técnicas relacionadas con el diseño

4.1. Los patrones de diseño software

Los patrones tienen su origen en el trabajo de C. Alexander en 1979 en un libro titulado *The Timeless Way of Building*, en el que se describen el lenguaje de patrones para que individuos o grupos de individuos construyan sus propias viviendas sin necesidad de arquitectos. C. Alexander afirma: "*cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta*

solución un millón de veces en el futuro sin tener que volver a pensarla otra vez".

Dentro de la ingeniería del software, Cunningham y Beck propusieron algunos patrones de interface de usuario a finales de los años 1980, pero su popularidad surgió con la obra de Gamma *et al* [13]. A día de hoy, se han especificado multitud de patrones en diferentes dominios y niveles de abstracción, por lo que en la industria del software los patrones se podrían clasificar acorde al ámbito que abarcan lo que se ha dado a llamar, *lenguajes de patrones*, como por ejemplo:

- Patrones de interfaces de usuario, interacción hombre-computador. El más conocido es quizás el MVC (Modelo-Vista-Controlador).
- Patrones de diseño, a nivel de un conjunto de clases en el paradigma de la orientación a objetos.
- Modismos (*programming idioms*), 'trucos' a nivel de lenguajes de programación concretos (C++, Java, etc.). Por ejemplo a veces son simples reglas a la hora de escribir código como prefijos para identificar el tipo de variables.
- Patrones para la integración de sistemas (EAI - *Enterprise Application Integration*), para la intercomunicación y coordinación de sistemas, por ejemplo para Java EE.
- Patrones a nivel de organización y flujos de trabajo, *workflows*, para la gestión de flujos de trabajo y procesos con sistemas empresariales, por ejemplo cómo organizar al personal y su forma de trabajar.

En particular, nos interesan los patrones de diseño muestran descripciones de soluciones a problemas a nivel de clases e interfaces, y las interacciones entre ellas.

Definición
Un patrón de diseño es una solución a un problema recurrente de carácter general a nivel de clases, interfaces, objetos y cómo estos interacturán.

Para que una solución sea considerada un patrón se considera que debe poseer ciertas características que lo validan. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares anteriormente, lo que se ha dado en llamar "*la regla de 3*"que afirma que algo es un patrón a menos que se haya usado en 3 ocasiones independientes. Otras son la comparación con otras posibles soluciones mostrando sus debilidades, que no hayan sido escritas por un único autor y que hayan sido criticadas por la comunidad. Aunque existe la barrera de aprendizaje de los patrones, una vez introducidos en los entornos de desarrollo, los patrones de diseño proporcionan las siguientes ventajas:

- La reutilización de arquitecturas completas yendo un paso más allá que la reutilización a nivel de código. Además representan conocimiento sobre decisiones de diseño.
- Formalizar un vocabulario común entre diseñadores.
- Facilitar el aprendizaje condensando conocimiento ya existente.

El libro de Gamma *et al* [13] también conocido como patrones GoF (*Gang of Four*, debido a sus 4 autores: Gamma, Helm, Johnson y Vlissides) describe 23 patrones concretos mediante una plantilla con los campos principales siendo su nombre, propósito, motivación, aplicabilidad, el desarrollo del patrón, usos y ejemplos. Además los patrones están clasificados en 3 categorías: (i) creacionales, (ii) estructurales y de (iii) comportamiento.

Como ejemplo de patrón, el patrón *Observer* define una dependencia de *uno a muchos* entre objetos, de modo que cuando un objeto cambia de estado todos los objetos dependientes son notificados y actualizados automáticamente. La *motivación* de este patrón sería la reducción del acoplamiento entre

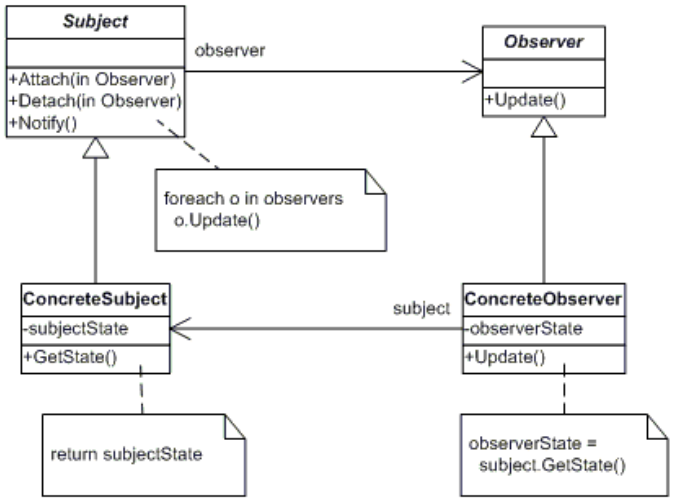


Figura 19: Patrón `Observer`

clase de manera que un objeto puede notificar a otros sin tener conocimiento de cuales objetos requieren la notificación. Un uso de este patrón se da en las interfaces de usuario, donde unos mismos datos pueden tener distintas vistas, y las distintas vistas son notificadas cuando los datos cambian. La estructura se pueden observar en diagrama de clases de la figura . En Java ya se tiene implementada la clase `Observer` y la interfaz `Observable`. La clase `java.util.Observable` es la clase abstracta *Subject* del patron y cualquier clase que quiera ser observada debe extender esta clase que proporciona métodos para añadir, borrar y modificar *observadores* que son guardados en un tipo Vector. Las subclases sólo son responsables de la llamada de notificación cuando el estado cambia, generalmente dentro de una método "`set...(...)`". Un bosquejo de código implementando el patrón con las clases proporcionadas en Java sería algo así:

```
...

interface Subject {
 public void addObserver(Observer o);
 public void removeObserver(Observer o);
 public String getState();
 public void setState(String state);
```

```
}

interface Observer {
 public void update(Subject o);
}

class ObserverConcreto implements Observer {
 private String state = "";

 public void update(Subject o) {
   estado = o.getState();
   System.out.println("Nuevo estado" + estado);
 }
}
```

Otros ejemplos de patrones muy conocidos incluyen el patron *Strategy*, también conocido como *Policy pattern*, que es utilizado para implementar distintos algoritmos que compartan la interfaz pero puedan utilizarse indistintamente, con la ventaja de que se puede cambiar de algoritmo dinámicamente (ver figura 20). El patrón *Composite* es utilizado para construir jerarquías de objetos que comparten una interfaz en la que unos elementos pueden formar parte de otros (por ejemplo, el las interfaces de usuario las ventanas se componen de paneles, botones campos de texto, etc. y a su vez por ejemplo los botones de una etiqueta de texto y una imagen). Otro patrón conocido es *Iterator* que nos permite recorrer los elementos de una clase contenedora de objetos (como puede ser un `Vector`) sin conocer su implementación; un ejemplo de su uso es la interface `java.util.Enumeration` como se muestra a continuación el el caso de tener una colección de `String`s:

```
Enumeration e = vector.elements();
while (e.hasMoreElements())
  String s = (String) e.nextElement();
  System.println(s);
}
```

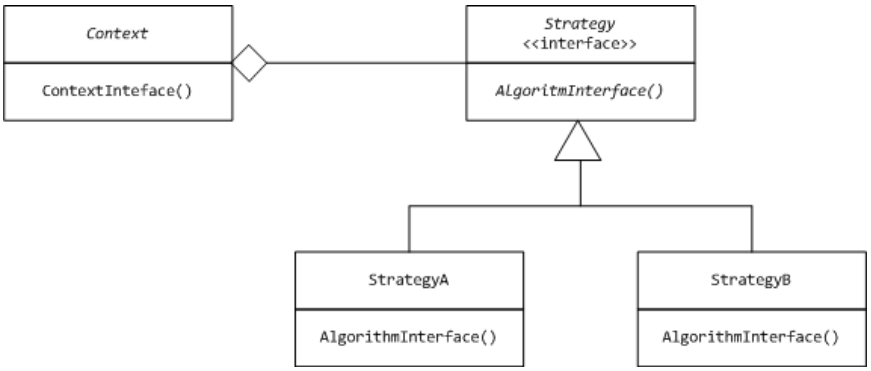


Figura 20: Patrón `Strategy`

Además, se puede hacer una reseña al concepto de *antipatrón*, que aunque se describen de manera semejante a la de un patrón, lo intentan es prevenir contra errores comunes en el diseño del software. Los antipatrones intentan detallar problemas que acarrean ciertos diseños muy comunes y que a primera vista correctos, pero que a la larga las desventajas superan los beneficios. Los antipatrones definen pueden definirse a distintos niveles, y por ejemplo, se han detallado antipatrones de gestión. Sin embargo, a nivel de diseño, los antipatrones puede clasificarse como *refactorizaciones*.

4.2. Software *frameworks*, *plug-ins* y componentes

Obviamente es mucho mejor la reutilización a nivel de arquitecturas y diseños que no a nivel de clases. Los patrones de diseño nos permiten reutilizar un pequeño número de clases en conjunto, los frameworks, plug-ins y componentes van más allá, permitiéndonos reutilizar diseños y arquitecturas enteras para dominios concretos.

4.2.1. *Frameworks*

Definición
Un *framework* es un conjunto de clases, interfaces y sus relaciones que proporcionan un diseño reusable para un sistema software o parte de un sistema (subsistema) de forma que sea extendido en una aplicación final.

Un *framework* es por tanto, un conjunto de clases y sus relaciones diseñado específicamente para ser extendido y no como aplicación final. Abstrae las entidades, estados y comportamientos en un dominio. Los *frameworks* proporcionan unos puntos de extensión, denominados *hot-spots* en inglés. La idea es proporcionar toda la funcionalidad genérica que se pueda y mostrar a los desarrolladores las interfaces (APIs – *Application Program Interface*) a extender para que sólo tengan que centrarse en lo específico de la aplicación. El ejemplo más típico y el origen de los *frameworks* está en las interfaces de usuario (GUIs), donde el usuario se centra en desarrollo de la interfaz abstrayéndose de cómo se implementan los sus distintos componentes, botones, ventanas, adornos, etc. Los *frameworks* no necesitan tener una parte visual, por ejemplo el *framework* para pruebas unitarias `JUnit`. Hoy día, los lenguajes de programación van acompañados de *frameworks* estándar proporcionados como parte del lenguaje, por ejemplo, Java con el JDK (Java Development Kit) o Java EE o C# con la plataforma .NET. Como ejemplo final, están surgiendo multitud de *frameworks* para el desarrollo de aplicaciones Web con los típica estructura de bases de datos y operaciones sobre ella, por ejemplo Apache Struts, Spring, etc. Generalmente, estos *frameworks* implementan un patrón muy general de alto nivel conocido como MVC (*Model-View-Controller*).

La motivación de desarrollar un *framework*, se da cuando este va a ser reutilizado en múltiples aplicaciones dándose una reducción de costes a largo plazo, ya que el desarrollo del *framework* en si mismo es mucho más costoso que el desarrollo de aplicaciones "normales". Además, en general, los *frameworks* tienen una larga curva de aprendizaje, por lo que en su diseño hay que llegar a un balance entra su simplicidad y la funcionalidad que proporciona.

Es decir, el *framework* debe ser lo suficientemente simple como para que pueda ser aprendido por los desarrolladores extendiendo sus *hot-spots*, pero además debe suministrar la funcionalidad suficiente para que *framework* sea útil en el domino para el que se ha diseñado.

4.2.2. *Plug-ins*

Además de los *frameworks*, las tendencias actuales para la mejora del la reutilización, está en la creación de *plug-ins*.

Definición
Un *plug-in* es una aplicación completa, módulo o componente que interacciona con una aplicación anfitriona (*host application*) extendiéndola o adaptando su comportamiento sin modificar la aplicación anfitriona.

Ejemplos de utilización diaria de plug-ins se dan el los navegadores Web, extendiendo su funciona básica de mostrar páginas en HTML o XML con terceras aplicaciones para ver documentos en pdf, videos, música, etc. Otro ejemplo se da en el desarrollo del software con plataformas genéricas, siendo *Eclipse*[2] y *Netbeans*[3] las más conocidas son, ya que permiten que terceras partes desarrollen aplicaciones con mucha de la funcionalidad ya proporcionada por estas plataformas y sin necesidad de modificarlas. Aparte de los aspectos técnicos, la no modificación de la plataforma anfitriona puede ser importante desde el punto de vista legal por el tema de licencias.

La forma de implementar *plug-ins* es mediante un patrón denominado Inversión de control (IoC – *Inversion of Control*), que se suele describir con el lema de Hollywood: *"No nos llame, nosotros le llamaremos"*. La idea es que la aplicación anfitriona no tenga que ser modificada, no tenga que conocer los detalles de implementación del *plug-in*.

[2] `http://www.eclipse.org/`
[3] `http://www.netbeans.org/`

4.2.3. Componentes

Definion – [29]
Los **componentes** son elementos de funcionalidad vendidos como una unidad e incorporados en múltiples usos. En la más pura forma de composición, los sistemas son construidos enteramente de componentes comprados (no localmente implementados) que son ensamblados sin modificación. El valor de los componentes se da cuando se ensamblan unos con otros componentes para formar un sistema como solución a medida.

La idea de los componentes de software se asemeja a la idea de componentes hardware con elementos intercambiables. Los componentes de software deben adherirse a una especificación que definen sus interfaces de tal manera que loas aplicaciones puedan construirse mediante la composición de interfaces o que terceras partes puedan desarrollar componentes sin tener que mostrar el código fuente. Ejemplos de componentes software son JavaBeans, Microsoft COM o componentes CORBA.

Esta forma de desarrollar sistemas ha dado lugar a la *ingeniería del software basada en componentes* (CBSE – *Component based Software Engineering*) donde se deben realizar una serie de tareas adicionales como puede ser la búsqueda, evaluación, selección, comprar y reemplazo de componentes y adaptar otras como pueden ser las pruebas. Esto ha dado lugar a que se adapten metodologías considerando 3 actividades particulares:

- *Cualificación de componentes*: actividades de análisis para evaluar cada componente.
- *Adaptación de componentes*: procedimientos para que los componentes se adapten a la arquitectura requerida.
- *Composición de componentes*: donde los ingenieros de software deben considerar los mecanismos de conexión y coordinación.
- *Actualización de componentes*: actividades para reemplazar compo-

nentes, donde la no disposición del código fuente, la compra a terceros y la dependencia de estos deben considerarse.

Diferencias entre bibliotecas de funciones, *frameworks*, *plug-ins* y componentes
▪ Las *bibliotecas de funciones* no tienen incorporado ningún flujo de control, se componen de clases o funciones que son llamadas para realizar una funcionalidad específica, por ejemplo, una biblioteca de funciones matemáticas. ▪ Los *frameworks* ya tienen diseñando una arquitectura y un flujo de control; están pensadas como aplicaciones incompletas diseñadas para ser extendidas. ▪ Los patrones de diseño se asemejar a los *frameworks*, pero desde el punto de vista lógico, y no de implementación como los *frameworks*. ▪ Los *componentes* están diseñados para ser piezas reemplazables, y la idea es desarrollar aplicaciones mediante composición. ▪ Los *plug-ins* se asemejan a los componentes, pero están diseñados para que sean llamados desde una aplicación anfitriona.

4.3. Diseño por contratos

Meyer [30] creo la noción de diseño por contratos (*Design by Contract*) en el contexto del la construcción de software orientado a objetos. Cada método tiene una precondición y una postcondición que esperan ser cumplidas respectivamente en la entrada y salida de la función. Las aserciones puede ser usadas para comprobar dichas condiciones una clase y sus clientes como

un acuerdo formal: "*si tú, cliente, me garantizas ciertas precondiciones, entonces yo, proveedor, generaré resultados correctos. Pero si de lo contrario, violas las precondiciones, no te prometo nada*".

La especificación de los contratos realiza por medio de aserciones como herramienta perfecta para expresar todos los derechos y obligaciones de cada una de las partes, del cliente con precondiciones, y las obligaciones del proveedor con postcondiciones.

Definición – [33]
Las aserciones son restricciones formales en el comportamiento de un sistema software que comúnmente están escritas como anotaciones en el código fuente.

Las *aserciones* fueron creadas por Hoare como un sistema de axiomas para demostrar la correctitud de programas Algol [17]. Además, dado que podemos expresar suposiciones del estado del programa durante la ejecución, podemos usar las aserciones como un modo de documentación, con la ventaja de que seremos notificados en el caso que nuestras suposiciones sobre el código no fueran ciertas. Siguiendo esta idea, podemos definir las especificaciones de un software OO, incluso antes de tener el código implementado, expresando sus precondiciones y poscondiciones de métodos. Además, como las aserciones son un modo de verificar nuestras suposiciones en cualquier punto del código, también podemos detectar errores de implementación. Podemos ahorrarnos una gran parte del tiempo de pruebas que gastaríamos buscando un error, ya que un fallo en la aserción nos devuelve información detallada sobre dicho error. Esta verificación tendría la forma de: {`Pre`} `Código` {`Post`}, donde un ejemplo muy trivial donde se prueba la correctitud de un trozo de código sería: {`a` $\leq$ `0`} `a++;` {`a` $\leq$ `1`}

5. Diseño de sistemas distribuidos

Los sistemas distribuidos se encuentran en todos los ámbitos de la vida, redes de ordenadores, sistemas de telefonía móvil, robots industriales, coches, aviones, trenes, etc. En esta sección se comenta brevemente las características de los sistemas distribuidos y como estas impactan en el diseño de sistemas y sus implementaciones. Además los sistemas distribuidos tienen conceptos y técnicas particulares.

Definición [6]
Un **sistema distribuido** es un sistema en el cual los components de hardware o software localizados en una red de ordenadores se comunican y coordinan sus acciones únicamente mediante el paso de mensajes.

Otra definición, quizás más sencilla, es la de Tanenbaum y van Steen [35]: *Ün sistema distribuido es una colección de ordenadores independientes que se comportan como un único ordenador de cara al usuario"*. La diferencia entre redes de computadores y los sistemas distribuidos es el nivel el que se encuentran. Mientras que las redes de computadores se centran el la interconexión y el envío de información por medio de paquetes, la rutas de los datos, direcciones IP, etc., los sistemas distribuidos muestran al usuario de forma transparente un conjunto de computadores trabajando de forma autónoma en una misma aplicación. Las motivaciones al crear sistemas distribuidos son varias:

- Funcional donde la distribución de la información es inherente al sistema (Web, correo, sistemas de compra-venta cliente/servidor, etc.)
- Distribución y balanceo de carga en servidores.
- Fiabilidad, donde se pueden tener sistemas redundantes y backups en diferentes localizaciones.
- Económicas, con la construcción de ordenadores potentes a base de

clusters de ordenadores más asequibles, o compartición de recursos como impresoras, backups, etc.

Un sistema distribuido puede ser desde un *cluster* de computadoras homogéneas en una red local y en una misma oficina hasta un sistema de ordenadores completamente heterogéneos (capacidades, sistemas operativos, etc.) distribuidos por distintos países y comunicándose a través de múltiples redes (wireless, Ethernet, ATM, etc.). Las consecuencias son que los sistemas distribuidos deben tener en cuenta incluyen:

Concurrencia Proporcionar y gestionar los accesos concurrentes a los recursos compartidos. En los sistemas distribuidos se debe gestionar ordenadores autónomos trabajando en paralelo y coordinando tareas teniendo en cuenta la no existencia de un "reloj" global en la comunicación, preservando las dependencias, evitando puntos muertos y proporcionando un acceso justo a los recursos.

Heterogeneidad Multiples tipos de ordenadores, redes y sistemas operativos que obliga a la creación de protocolos abiertos para su permitir su comunicación por paso de mensajes.

Transparencia La distribución debería ocultarse de los usuarios y a los desarrolladores. Hay diferentes tipos:

- *Transparencia en el acceso*: el acceso a ficheros locales o remotos debería ser idéntico (p.e., sistemas de ficheros en red).
- *Transparencia en la localización*: la aplicación distribuida debería de permitir el acceso a los recursos independientemente de donde se encuentren (p.e., los servicios Web o CORBA pueden dinámicamente descubrir e invocar servicios independientemente de localización).

Tolerancia a fallos Los sistemas distribuidos necesitan recuperarse de forma transparente a fallos en cualquiera de los componentes de un sistema

distribuido. Para ello deben de implementar mecanismos de detección de fallos, enmascaramiento de fallos, tratamiento de excepciones, recuperación de fallos con mecanismos de *rollback*, etc.

Escalabilidad Los sistemas distribuidos deberían funcionar eficientemente cuando se incrementa el número de usuarios o escalar la eficiencia acorde al número de recursos nuevos que se ponen en la red.

Seguridad El sistema distribuido debería de ser usado solamente de la manera que se diseño para mantener la confidencialidad, que individuos no accedan a información no autorizada, integridad, protección contra la alteración o corrupción de información, disponibilidad contra ataques y el no-repudio mediante pruebas de envío y recepción de la información. Esto se consigue mediante los mecanismos de encriptación, autentificación y autorización.

Dentro de los estilos arquitecturales que hemos comentado, los que se pueden englobar dentro de los sistemas distribuidos incluyen cliente-servidor, *peer-to-peer* y *grid*. Existe una variedad de tecnologías para implementar dichos estilos arquitecturales entre los que se encuentran:

- Un **servicio Web** es un sistema software identificado por una URI (Uniform Resource Identifier), cuyas interfaces públicas y puntos de enlace (*bindings*) están definidos y descritos en XML. Su definición puede ser descubierta por sistemas software. Estos sistemas pueden entonces interactuar con el servicio Web de la forma prescrita en su definición, utilizando mensajes basados en XML llevados a cabo por protocolos de Internet.
- **CORBA** (Common Request Broker Architecture) es un *middleware* para la programación concurrente de sistemas distribuidos mediante la orientación a objetos. CORBA proporciona una plataforma para invocar objetos en servidores remotos, que pueden encontrarse en la misma maquina o a través de redes. Utilizando CORBA, una clase

puede transparentemente invocar a métodos de otras clases sin saber su localización (en qué máquina se encuentra dicha clase).

- Otras tecnologías que permiten implementar sistemas distribuidos aunque específicas a lenguajes de programación o plataformas se encuentran **RMI** (*Remote Method Invocation*) en Java o **DCOM** (*Distributed Component Object Model*) en Microsoft Windows.

6. Evaluación y métricas en el diseño

Los diseños necesitan ser evaluados de forma que puedan evaluarse distintos criterios de calidad y comparar distintas alternativas, ya que dada una especificación, pueden existir varios diseños perfectamente válidos pero con distintos criterios de calidad. Entre los criterios más importantes:

- *Extensibilidad*. La capacidad de añadir nueva funcionalidad sin necesidad de cambios significativos en su arquitectura.
- *Solidez*. la capacidad de operar bajo presión y tolerar entradas invalidas o impredecibles.
- *Fiabilidad*. la capacidad de llevar a cabo la función requerida en las condiciones deseadas durante un tiempo especificado.
- *Tolerancia a fallos*. el sistema debe robusto y capaz de recuperarse ante fallos.
- *Compatibilidad*. La capacidad de operar con otros productos y su adherencia a estándares de interoperabilidad.
- Reusabilidad. Relacionado con el concepto de modularidad, los componentes deberían de capturar la funcionalidad esperada, ni más ni menos y adherirse . Esta única finalidad, hace que los componentes sean reutilizables en otros diseños con idénticas necesidades.

Un tipo de validación, aunque menos común, es la validación matemática. Los diseños se pueden validar matemáticamente si se ha seguido un una metodología formal en la especificación de requisitos. Si en el diseño de la arquitectura se ha especificado utilizando algún lenguaje formal de definición de arquitectura, pueden validarse y compararse distintas alternativas mediante herramientas especificas antes de su implementación definitiva.

Otras técnicas más comunes son las revisiones e inspecciones, que al igual que todas las demás fases del ciclo de vida, se pueden realizar unas reuniones llamadas revisiones o inspecciones de diseño donde se analiza los artefactos que son generados en la fase correspondiente. Las revisiones de diseño se suelen clasificar según el nivel de detalle en:

- Revisiones de diseño preliminares, donde se examinan el diseño conceptual con los clientes y usuarios.
- Revisiones de diseño críticas, donde el diseño se presenta a los desarrolladores.
- Revisiones del diseño del programa, donde los programadores obtienen *feedback* antes de la implementación. A estas reuniones también se las conoce como inspecciones (o *walkthroughs* en si el moderador de la reunión es la persona que ha generado el diseño o programa, siendo generalmente más informales.

En la sección 2.3 se han descritos los conceptos de cohesión y acoplamiento, siendo el objetivo más importante del diseño el maximizar la cohesión y minimizar el acoplamiento. Además, en esta sección veremos algunos ejemplos de métricas relacionadas con la complejidad del diseño, prácticamente sinónimo de la complejidad estructural.

La medición de la complejidad estructural se realiza midiendo las llamadas entre módulos con los conceptos de *fan-in* y *fan-out* (ver figura 21):

- ***Fan-in***, o grado de dependencia, de un módulo es número de módulos que llaman a dicho módulo.

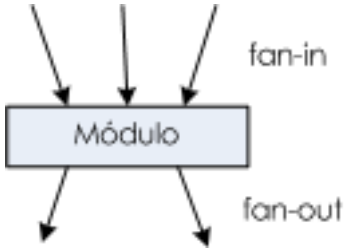


Figura 21: Conceptos de *fan-in* y *fan-out*

- ***Fan-out***, o grado de responsabilidad de coordinación, de un módulo es el número de módulos que son llamados por dicho módulo.

Un valor alto de *fan-in* indica que el módulo está fuertemente acoplado por lo que cambios en el módulo afectarán resto del sistema. Valores altos de *fan-out* pueden indicar módulos complejos debido a la complejidad de la lógica de control necesaria para coordinar las llamadas al módulos.

Basándose en estos conceptos de flujo de información, Henry y Kafura [16] una conocida métrica de la **complejidad estructural**:

$$HK_m = C_m \cdot (fan - in_m \cdot fan - out_m)^2$$

donde C_m es la complejidad del módulo generalmente medida en LoC, aunque pueden considerarse otras como la complejidad ciclomática (ver **??**)

Basándose en teoría de grafos, Fenton [11] propone medidas de morfología de la estructura de módulos jerárquicos del sistema:

- Tamaño = número de nodos + número de aristas
- Profundidad de anidamiento
- Anchura
- Proporción de arcos y nodos, número de arcos / número de nodos

Dentro de la orientación a objetos, entre las métricas más conocidas son las conocidas como MOOD (*Metrics for Object oriented Design*) definidas por Brito e Abreu y Melo [2] y las de Chidamber y Kemerer [5]. Estas métricas, en líneas generales, identifican clases mal diseñadas mediante mediciones de mecanismos estructurales básicos en el paradigma de la orientación a objetos como encapsulación, herencia, polimorfismo y paso de mensajes. Además, las métricas a nivel de sistema, y por ende si el sistema en general está bien diseñado, pueden derivarse de las métricas de a nivel de clase métricas usando la media u otros estadísticos. Entre las métricas MOOD podemos citar como ejemplos:

- **Proporción de métodos ocultos** (MHF – *Method Hiding Factor). Se define como la proporción del número de métodos definidos como* protegidos o privados entre el total de métodos. Esta métrica mide la encapsulación.
- **Proporción de atributos ocultos** (AHF – *Attribute Hiding Factor*) Es la proporción entre los atributos definidos como protegidos o privados y el número total de atributos. Aunque a veces por mejorar el rendimiento se acceden o modifican los atributos directamente, idealmente esta métrica debería ser 100 %, i.e., se deberían de tener todos los atributos privados y acceder a ellos mediante métodos *get/set*. Esta métrica también mide la encapsulación.
- **Proporción de métodos heredados** (MIF – *Method Inheritance Factor*). Se define como la proporción de la suma de todos los métodos heredados en todas las clases entre el número total de métodos (localmente definidos más los heredados) en todas las clases.
- **Proporción de atributos heredados** (AIF– *Attribute Inheritance Factor*). Se define como la proporción del número de atributos heredados entre el número total de atributos. AIF se considera un medio para expresar el nivel de reusabilidad en un sistema.
- **Proporción de polimorfismo** (PF – P*olymorphism Factor*). PF se

define como la proporción entre el número real de posibles diferentes situaciones polimórficas para una clase C_i entre el máximo número posible de situaciones polifórficas en C_i. En otras palabras, el número de métodos heredados redefinidos dividido entre el máximo número de situaciones polimórficas distintas. PF es una medida del polimorfismo y una medida indirecta de la asociación dinámica en un sistema.

Al igual que las métricas MOOD, Chidamber y Kemerer [5] han definido una serie de métricas que han sido ampliamente adoptadas para medir características como encapsulamiento, ocultamiento de información, herencia. Entre estas métricas tenemos:

- **Acoplamiento entre objectos** (CBO – *Coupling Between Objects*). CBO de una clase es el número de clases a las cuales está ligada, es decir, usa métodos o variables de otra clase (medida del *fan-out*). Las clases relacionadas por herencia no se tienen en cuenta. Sistemas en los cuales una clase tiene un alto CBO y todas las demás tienen valores próximos a cero indican una estructura no orientada a objetos, con una clase principal dirigente. Por el contrario, la existencia de muchas clases con un alto valor de CBO indica que el diseñador ha afinado demasiado la "granularidad" del sistema. Esta métrica además puede utilizarse para medir el esfuerzo en el mantenimiento y las pruebas. A mayor acoplamiento, mayor dificultad de comprensión y reuso, mantenimiento en acoplamiento se da en una clase, más difícil sera reutilizarla. Además, las clases con excesivo acoplamiento dificultan la comprensibilidad y hacen más difícil el mantenimiento por lo que será necesario un mayor esfuerzo y rigurosas pruebas.

- **Respuesta para una clase** (RFC – *Response For a Class*) Esta métrica cuenta las ocurrencias de llamadas a otras clases desde una clase particular y mide tanto la comunicación interna como la externa. RFC es una medida de la complejidad de una clase a través del número de métodos y de la comunicación con otras clases.

- **Profundidad en árbol de herencia** (DIT – *Depth of Inheritance Tree*). Mide el máximo nivel en la jerarquía de herencia. DIT es la cuenta directa de los niveles en la jerarquía de herencia. En el nivel cero de la jerarquía se encuentra la clase raíz. A medida que crece su valor, es más probable que las clases de niveles inferiores hereden muchos métodos y esto da lugar a posibles dificultades cuando se intenta predecir el comportamiento de una clase y por lo tanto, su mantenimiento. Una jerarquía profunda conlleva también una mayor complejidad de diseño. Por otro lado, los valores grandes de DIT implican que se pueden reutilizar muchos métodos, lo que debe ser considerado como un elemento a favor de la mantenibilidad.

- **Número de descendientes** (NOC – *Number of Children*). Cuenta el número de subclases subordinadas a una clase en la jerarquía, es decir, el número de subclases que pertenecen a una clase. A medida que crece el número de descendientes, se incrementa la reutilización, pero también implica que la abstracción representada por la clase predecesora se ve diluida. Esto dificulta el mantenimiento, ya que existe la posibilidad de que algunos de los descendientes no sean realmente miembros propios de la clase.

- **Métodos ponderados por clase** (WMC – *Weighted Methods per Class*) Dado un peso a cada método en una clase, WMC es el sumatorio todos los pesos de la clase. Se supone que WMC mide la complejidad de una clase, pero si todos los métodos son considerados igualmente complejos (mismo peso), entonces WMS es simplemente el número de métodos definidos en una clase.

- **Falta de cohesión en los métodos** (LCOM – *Lack of Cohesion in Methods*). LCOM establece en qué medida los métodos hacen referencia a atributos. LCOM es una medida de la cohesión de una clase midiendo el número de atributos comunes usados por diferentes métodos, indicando la calidad de la abstracción hecha en la clase.

7. Resumen

Hemos definido los conceptos fundamentales de diseño y proporcionado una visión global de los mecanismos y pasos para su realización. Mediante la abstracción, descomposición y modularización, primero se lleva a cabo el *diseño de alto nivel* para después refinarlo en en lo que se conoce como *diseño detallado* de los componentes e interfaces que componen un sistema.

Según han ido evolucionando los lenguajes de programación (no estructurados, estructurados, orientados a objeto) han ido evolucionando las notaciones y técnicas de diseño. Por ejemplo, si bien hoy día es común que un mismo proyecto utilize distintas notaciones, los diagramas de flujo de datos son típicos de los métodos estructurados, los diagramas de clase, de la programación orientadas a objetos, y para el diseño de la base de datos es común utilizar modelos entidad/relación. Por sus peculiaridades, se han comentado aparte el diseño de los sistemas distribuidos y en tiempo real. Además, hemos cubierto brevemente otras técnicas relacionadas con el diseño como son los patrones de diseño, la *refactorización*, *frameworks*, *plug-ins* y el diseño por contratos.

Finalmente, se descrito la los criterios de calidad y técnicas de evaluación del diseño principalmente mediate inspecciones y métricas relevantes al diseño, principalmente como medida de la modularización mediante los conceptos de acoplamiento y cohesión.

8. Notas bibliográficas

Existen literalmente cientos y excelentes de libros cubriendo los aspectos de diseño tratados en este capitulo tanto en español como en inglés y aunque se intentarán incluir los trabajos seminales y más relevantes siempre se quedarán excelentes obras sin mencionar.

Entre los libros cubriendo exclusivamente la fase de diseño, Budgen [4] destaca la importancia del diseño en el contexto global del desarrollo del

software y cubre tanto las metodologías estructuradas como las orientadas a objetos (aunque resalta más los métodos estructurados y metodologías tradicionales).

Sobre el diseño utilizando métodos estructurados, las referencias son las de sus autores originales como [14], [19], [39], etc. comentadas en el texto pero muchas de ellas resumidas en el libro de Budgen [4]. En cuanto al diseño orientación a objetos, existen literalmente cientos de excelentes referencias siendo de obligada cita la de B. Meyer [30]. Otras de más reciente publicación y que cubren la ingeniería del software desde el punto de vista de la orientación a objetos podríamos citar a Bruegge y Dutoit [3]. Además dentro de la orientación a objetos, sobre UML y el proceso unificado son de referencia obligatoria las obras de sus autores originales. G. Booch [1], J. Rumbaugh [34] e I. Jacobson [20]. Entre ellos han escrito tres volúmenes: una guía del lenguaje UML, una referencia de UML y el proceso unificado. Otras referencia importante por su sencillez y rápida lectura como introducción a UML es la de M. Fowler y Kendall [12]. La referencia recomendada de OCL es la de Warmer y Kleppe [37]. También de los últimos autores junto con Bast, es muy recomendable su trabajo sobre introducción a la arquitectura dirigida por modelos, MDA [22].

El libro de patrones por excelencia y principal fuente de referencia es el de Gamma *et al* [13]. Otro más recientemente publicado es Larman [24]. Además existen multitud de sitios Web explicando los patrones principales en diferentes lenguajes de programación. Relacionado con patrones y refactorización la referencia es el libro de Fowler [12].

Los trabajos originales describiendo *frameworks* son los de Johnson y Foote [21], aunque un libro más reciente es de Fayad et al [10].

Dos libros de obligada referencia sobre los sistemas distribuidos son los de Coulouris *et al* [6] y el de Tanenbaum y van Steen [35]. Ambos libros de docencia cubren todos los aspectos relacionados con los sistemas distribuidos y no solamente el aspecto de diseño.

Finalmente, en cuanto a las métricas y evaluación, nos remitimos a la

referencia principal del capítulo de medición [11], y a los artículos citados en el texto [5] y [2].

Referencias

[1] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, 2nd Edition.* Addison-Wesley Professional, 2005.

[2] 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*, pages 90–99, Washington, DC, USA, 1996. IEEE Computer Society.

[3] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering: Using UML, Patterns and Java, Second Edition.* Prentice Hall, September 2003.

[4] David Budgen. *Software Design.* Addison Wesley, 2003.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[6] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems (3rd ed.): concepts and design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[7] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming.* Academic Press Ltd., London, UK, UK, 1972.

[8] Tom DeMarco. Structured analysis and system specification. 1979.

[9] Edsger W. Dijkstra. Notes on structured programming. Technical Report 70-WSK-03, Technological University Eindhoven, 1970.

[10] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[11] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: a Rigorous & Practical Approach*. International Thompson Press, 1997.

[12] Martin Fowler. *UML Distilled: A Brief Guide to the Standard*. Adison-Wesley, 2004.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] Chris P. Gane and Trish Sarson.

[15] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[16] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 26(1):53–56, 1983.

[18] IEEE. IEEE standard glossary of software engineering terminology, 1990.

[19] M.A. Jackson. *Principles of Program Design*. Academic Press, Inc., Orlando, FL, USA, 1975.

[20] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[21] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[22] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[23] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[24] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[25] Barbara Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5), May 1988.

[26] Karl J. Lieberherr Ian M. and Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.

[27] R. C. Martin. The Dependency Inversion Principle. *C++ Report*, 8(6):61–66, June 1996.

[28] Robert C. Martin. Object oriented design quality metrics: An analysis of dependencies. *ROAD*, 2(3), Sep-Oct 1995.

[29] D. Messerschmitt. Rethinking components: From hardware and software to systems. *Proceedings of the IEEE*, 95(7):1473–1496, July 2007.

[30] Bertarnd Meyer. *Construcción de software Orientación a objetos.* Prentice-Hall, 1999.

[31] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of ACM*, 15(12):1053–1058, 1972.

[32] R. Pressman. *Ingeniería del Software.* McGraw–Hill, 2001.

[33] David S. Rosenblum. A practical approach to programming with assertions. 21(1):19–31, 1995.

[34] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, 2nd Edition*. Pearson Higher Education, 2004.

[35] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. 2nd edition edition, 2006.

[36] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[37] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[38] Jean Dominique Warnier. *Logical Construction of Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1981.

[39] Edward Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1993.