

Tema 1. Desarrollo utilizando patrones de diseño y arquitectónicos

Desarrollo de Software

Curso 2022-2023

3º - Grado Ingeniería Informática

Dpto. Lenguajes y Sistemas Informáticos

ETSIIT

Universidad de Granada

11 de mayo de 2023



Tema 1. Desarrollo utilizando patrones de diseño y arquitectónicos

Contenidos

1.1.	Análisis y diseño basado en patrones	5
1.1.1.	Origen e historia de los patrones software	5
1.1.2.	Conceptos generales y clasificación	8
1.1.3.	Relación con el concepto de marco de trabajo (framework)	11
1.1.4.	Elementos de un patrón de diseño	12
1.1.5.	Un ejemplo práctico a partir de las clases Modelo-Vista-Controlador (MVC)	13
1.2.	Cómo resolver problemas de diseño usando patrones de diseño	17
1.3.	Estudio del catálogo GoF de patrones de diseño	19
1.3.1.	Patrones creacionales <i>Factoría Abstracta</i> , <i>Método Factoría</i> , <i>Prototipo</i> y <i>Builder</i>	20
1.3.2.	Patrones estructurales <i>Facade</i> , <i>Composite</i> , <i>Decorator</i> y <i>Adapter</i>	37
1.3.3.	Patrones conductuales <i>Observer</i> , <i>Visitor</i> , <i>Strategy</i> , <i>TemplateMethod</i> e <i>InterceptingFilter</i>	47
1.4.	Patrones o estilos arquitectónicos	54
1.4.1.	Estilos de Flujo de Datos: el estilo <i>Tubería y filtro</i>	55
1.4.2.	El estilo <i>Abstracción de Datos y Organización OO</i>	57
1.4.3.	El estilo <i>Modelo-Vista-Controlador</i> (MVC)	59
1.4.4.	El estilo <i>Basado en Eventos</i>	64
1.4.5.	El estilo <i>Sistema por Capas</i>	67
1.4.6.	Estilos Centrados en Datos. El estilo <i>Repositorio</i>	69
1.4.7.	Estilos de Código Móvil. El estilo <i>Intérprete</i>	71
1.4.8.	Estilos heterogéneos. Estilo <i>Control de procesos</i>	72
1.4.9.	Otros estilos arquitectónicos	77
1.4.10.	Combinación de estilos y frontera débil con patrones de diseño	85
	Acrónimos	87
	Bibliografía	89

Tema 1. Desarrollo utilizando patrones de diseño y arquitectónicos

Contenidos

4

Desarrollo utilizando patrones de diseño y arquitectónicos

En este tema estudiaremos el concepto de patrón software y sus tipos (sección [1.1](#)) y la forma de resolver problemas de diseño (sección [1.2](#) para pasar después a estudiar con detalle patrones de diseño (sección [1.3](#)) y, por último, patrones o estilos arquitectónicos (sección [1.4](#)).

1.1. Análisis y diseño basado en patrones

1.1.1. Origen e historia de los patrones software

Los patrones software fueron la adaptación al mundo de las Tecnologías de la Información y de la Comunicación (TICs) de los patrones arquitectónicos, que fueron definidos en 1966 por el arquitecto y teórico del diseño, Christopher Alexander (figura [1.1](#)) (estadounidense nacido en Austria) como la identificación de ideas de diseño arquitectónico mediante descripciones arquetípicas y reusables. Sus teorías sobre la naturaleza del diseño centrado en el hombre han repercutido en otros campos como en la sociología y en la ingeniería informática.



Figura 1.1: Christopher Alexander. Arquitecto que define el concepto de patrón arquitectónico.

En 1987 se adaptaron al desarrollo de software y se presentó la idea en un congreso.¹

El primer libro sobre patrones software se publicó en 1994 por la llamada “Gang of Four”,² con una versión en CD³ y una nueva edición en 1995.⁴ En él se explica cómo el concepto de patrón software es una adaptación directa del concepto de patrón usado por los arquitectos, que es definido como:

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.⁵

Se trata de no “reinventar la rueda”, tampoco en ingeniería del software:

One thing expert designers know not to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is

¹Kent Beck y Ward Cunningham, «Using Pattern Languages for Object Oriented Programs», en *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1987), <http://c2.com/doc/oopsla87.html>.

²Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (USA: Addison-Wesley Longman Publishing Co., Inc., 1994), <https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/>.

³Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* (USA: Addison-Wesley Longman Publishing Co., Inc., 1994).

⁴Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (USA: Addison-Wesley Longman Publishing Co., Inc., 1995).

⁵Brad Appleton, *Patterns and Software: Essential Concepts and Terminology*, 2000, <http://www.brada.com/docs/patterns-intro.html>.



Figura 1.2: El lugar donde se creó el “HillSide Group”, en las montañas de Colorado en 1993.

part of what makes them experts. Consequently, you’ll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.⁶

Un año antes de la publicación de este libro, y ya conocidas las ideas de Erich Gamma y su “banda”, Kent Beck, junto con Grady Booch (uno de los creadores del lenguaje UML, junto con Rumbaugh y Jacobson) organizaron un retiro en las montañas del Colorado al que fueron expertos en desarrollo de software para intentar casar las ideas de patrón arquitectónico con la de objeto software, a la manera en que lo hicieron la GoF pero intentando que el patrón recogiera la idea original de creatividad de los patrones arquitectónicos de Christopher Alexander. Como se alojaron en la loma de una colina (1.2), al grupo que crearon le llamaron el “HillSide Group” ([The HillSide Group](#)).

En la actualidad se considera una ONG educativa y organiza congresos relacionados con los patrones software, como por ejemplo la [European Conference on Pattern Languages of Programs \(EuroPLoP\)](#) y mantienen catálogos de patrones y editan libros relacionados.

En 1996 se publicó otro libro de patrones software, esta vez más general, con el apoyo del “HillSide group”. Si el primero era solo sobre patrones de diseño (nivel medio), éste abarcaba desde los patrones de alto nivel o arquitectónicos hasta los llamados patrones a nivel de código (idioms) o patrones de bajo nivel.⁷ A veces al grupo de cinco autores que lo publicó se le ha llamado la “Gang of Five” por similitud con la GoF.

⁶Gamma et al., [Design Patterns: Elements of Reusable Object-Oriented Software](#).

⁷Frank Buschmann et al., [Pattern-Oriented Software Architecture - Volume 1: A System of Patterns](#) (Wiley Publishing, 1996), <https://learning.oreilly.com/library/view/pattern-oriented-software-architecture/9781118725269/>.

1.1.2. Conceptos generales y clasificación

Definición de patrón El patrón software se ha definido de la siguiente forma:

A pattern involves a general description of a recurring solution to a recurring problem replete with various goals and constraints. But a pattern does more than just identify a solution, it also explains why the solution is needed!.⁸

Otra definición es:

A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.⁹

La GoF, siendo los primeros en hablar de patrones, se enfocaron de forma específica en los patrones de diseño, haciendo un catálogo de ellos que veremos más adelante. Pero ellos mismos ya consideraban que existen otros tipos de patrones y daban algunos ejemplos de ellos:

- patterns dealing with concurrency or distributed programming or real-time programming ...
- application domain-specific patterns ...
- how to build user interfaces,
- how to write device drivers, or
- how to use an object-oriented database.

Each of these areas has its own patterns, and it would be worthwhile for someone to catalog those too..¹⁰

Clasificación general de los patrones Los patrones surgen desde la orientación a objetos y resuelven problemas a nivel de diseño orientado a objetos:

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not it can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample C++

⁸Appleton, *Patterns and Software: Essential Concepts and Terminology*.

⁹Buschmann et al., *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*.

¹⁰Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*.

and (sometimes) Smalltalk code to illustrate an implementation. Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages¹¹

Pero enseguida se amplían a cualquier paradigma de programación, básicamente eliminando los términos “orientado a objetos” y cambiando los términos de clase por módulo o subsistema. Además se conciben para aportar soluciones en un espectro mucho más amplio en el nivel de abstracción. Se pasa de concebirse únicamente como soluciones de diseño (los llamados patrones de diseño, que están en un nivel medio de abstracción) a concebirse como soluciones en cualquier nivel desde el más abstracto o genérico (los patrones arquitectónicos, en el nivel arquitectónico) hasta el más específico (los patrones de código o expresiones liguísticas, “idioms” en inglés):¹²

- Architectural Patterns.- An architectural pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- Design Patterns.- A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.
- Idioms.- An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Otra clasificación de los patrones los divide según la fase dentro del ciclo de vida de desarrollo del software, en patrones conceptuales, de diseño y de programación:¹³

- *Conceptual Patterns*.- A conceptual pattern is a pattern whose form is described by means of terms and concepts from an application domain.
- *Design Patterns*.- A design pattern is a pattern whose form is described by means of software design constructs, for example objects, classes, inheritance, aggregation and use-relationship.
- *Programming Patterns*.- A programming pattern is a pattern whose form is described by means of programming language constructs.

¹¹Appleton, *Patterns and Software: Essential Concepts and Terminology*.

¹²Appleton, *Patterns and Software: Essential Concepts and Terminology*; Buschmann et al., *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*.

¹³Appleton, *Patterns and Software: Essential Concepts and Terminology*.

Clasificación de los patrones de diseño La GoF subdivide los patrones de diseño en base a dos criterios. El primero, es el propósito, y según él, establecen tres tipos de patrones de diseño¹⁴ (pp 21 y 22):

- Creacionales.- Relacionados con el proceso de creación de objetos.
- Estructurales o estáticos.- Relacionados con los componentes que forman las clases y los objetos.
- Conductuales o dinámicos.- Relacionados con la forma en la que los objetos y las clases interactúan entre sí y se reparten las responsabilidades.

El segundo criterio es el ámbito de aplicación, y según él, hay dos tipos de patrones de diseño:

- De clase.- El patrón se aplica principalmente a clases.
- De objeto.- El patrón se aplica principalmente a objetos.

La Tabla 1.1 muestra ejemplos de patrones de diseño en base a estos dos criterios de clasificación.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Tabla 1.1: Espacio de los patrones de diseño [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], pp. 21-22)].

Otra forma en la que los patrones de diseño son comprendidos, es mediante las semejanzas entre ellos. La GoF las representa usando un grafo dirigido, tal y como se muestra en la figura 1.3¹⁵ (p. 23).

¹⁴Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*.

¹⁵Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD*.

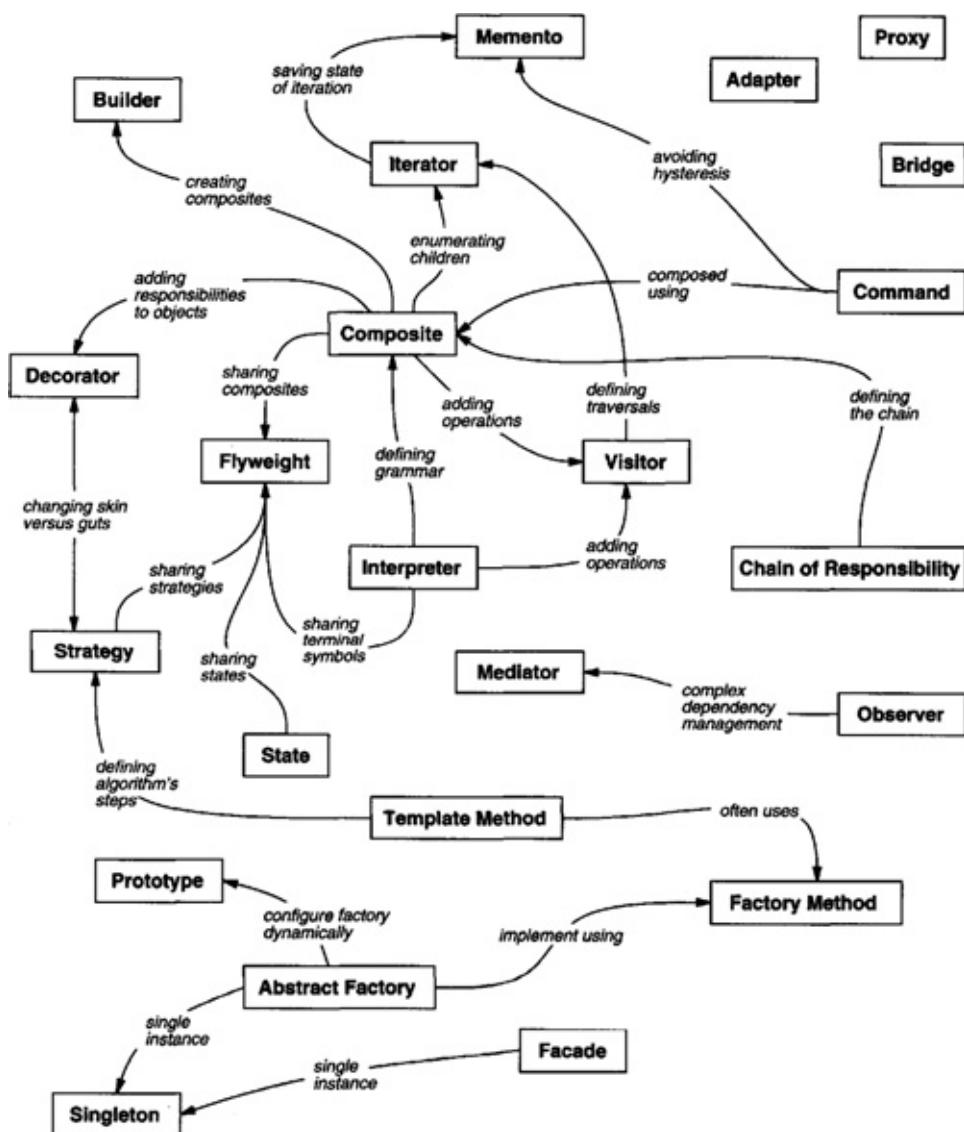


Figura 1.3: Relación entre los patrones de diseño. [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 23]

1.1.3. Relación con el concepto de marco de trabajo (framework)

Por otro lado, hay una cierta relación entre el concepto de patrón de diseño y el concepto de marco de trabajo (framework), pero nunca deben ser confundidos:

- **Marco de trabajo.-** Un conjunto amplio de funcionalidad software ya implementada que es útil en un dominio de aplicación específico, tal como un sistema de gestión de bases de datos, un tipo de aplicaciones web, etc.

- Patrón software (de diseño, arquitectónico ...).- NO ESTÁ IMPLEMENTADO; es una guía, una “receta”, una prescripción de desarrollo software, aplicable en cualquier dominio de aplicaciones, capaz de dar la misma solución a distintos problemas con una base similar, haciendo una abstracción de la parte común de los mismos que es crucial para llegar a una solución que simplifique la implementación y reusabilidad del código.

1.1.4. Elementos de un patrón de diseño

La GoF identificó cuatro elementos esenciales en un patrón de diseño:¹⁶

1. The pattern **name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.
2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

¹⁶Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*.

En la actualidad se han identificado más elementos y se utiliza una plantilla (Tabla 1.2) con todos esos elementos cuando se proporciona un patrón (en negrita los cuatro elementos principales identificados por la GoF):

Plantilla	
Nombre	el nombre dado al patrón
Clasificación	arquitectónico/de diseño/de programación
Contexto	describe el entorno en el que se ubica el problema incluyendo el dominio de aplicación
Problema	una o dos frases que explican lo que se pretende resolver
Consecuencias	lista el sistema de fortalezas que afectan a la manera en que ha de resolverse el problema; incluye las limitaciones y restricciones que han de respetarse
Solución	proporciona una descripción detallada de la solución propuesta para el problema
Intención	describe el patrón y lo que hace
Anti-patrones	“soluciones” que no funcionan en el contexto o que son peores; suelen ser errores cometidos por principiantes
Patrones relacionados	referencias cruzadas relacionadas con los patrones de diseño
Referencias	reconocimientos a aquellos desarrolladores que desarrollaron o inspiraron el patrón que se propone
Estructura	Diagrama UML
participantes	descripción de los componentes (clases/objetos) que lo forman y su papel

Tabla 1.2: Plantilla utilizada para describir un patrón.

1.1.5. Un ejemplo práctico a partir de las clases Modelo-Vista-Controlador (MVC)

La GoF pone un ejemplo práctico de uso de patrones a partir de tres clases en Smalltalk para construir interfaces gráficas de usuario (Graphical User Interfaces, GUIs)¹⁷. En otros lenguajes pueden implementarse GUIs usando el mismo diseño, o variantes. Así por ejemplo, en Java podemos usar más de una clase, agrupada en un paquete, para el modelo, otro para la vista (salida) y otro para el controlador (captura la petición del usuario y la pasa al modelo y/o la vista). Pero la idea es siempre la misma, separar el objeto de la aplicación (el modelo) de la interacción con el usuario (entradas y salidas). En otros diseños, por ejemplo el que utiliza el paquete Java SWING de diseño de GUIs, controlador y vista se mantienen unidos en lo que se llama modelo de gestión de eventos (event handling modelling). En todo caso, siempre se trata de separar el modelo de su presentación e interacción con él,

¹⁷Estas clases forman precisamente otro patrón, pero a nivel arquitectónico, y por aquél entonces aún no se había identificado como tal.

para aumentar su flexibilidad y reusabilidad. Así, podemos decidir poner diferentes GUIs, o hacer incluso una modalidad web o una app para el móvil, y el modelo quedaría siempre intacto. Debe ser además válido en configuraciones multiusuario donde el mismo modelo es accesible simultáneamente por varios usuarios, con distintas modalidades de interfaces. Android Studio, como ejemplo de entorno integrado para desarrollo de aplicaciones móviles, ha incorporado también el modelo tripartito **Modelo-Vista-Controlador (MVC)** de diseño arquitectónico de aplicaciones gráficas. Y lo mismo ocurre con el framework para aplicaciones web Ruby on Rails. La GoF utiliza el diseño **MVC** para identificar e introducir los tres primeros patrones de su libro,¹⁸ que se refieren a continuación:

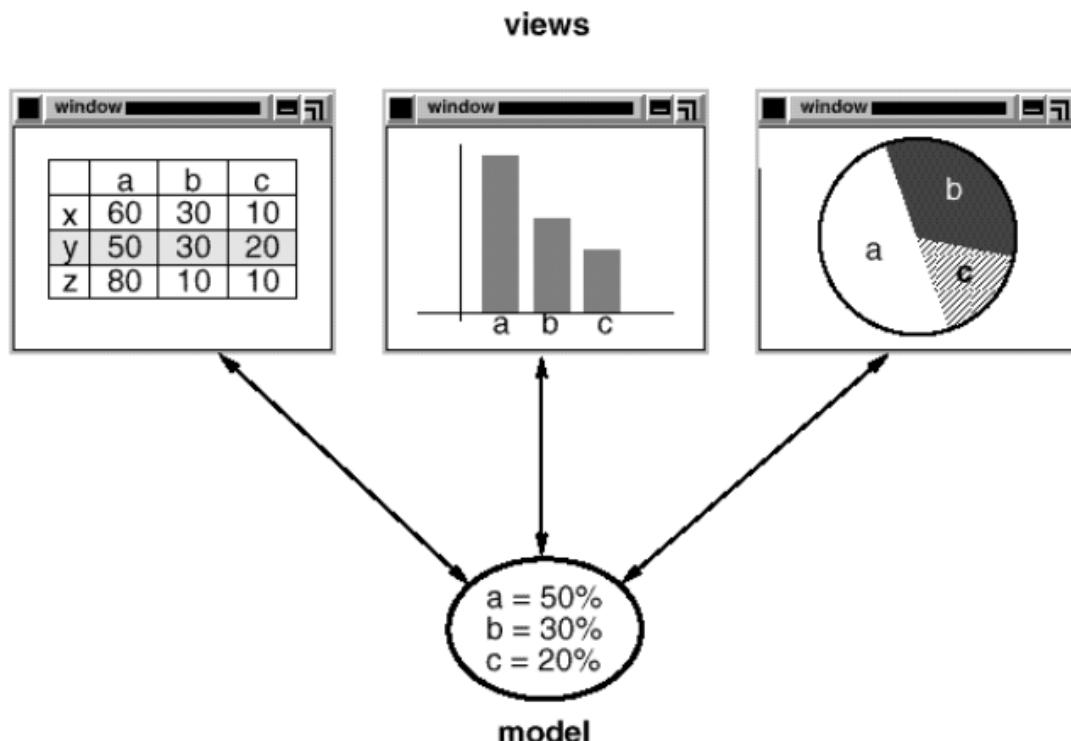


Figura 1.4: Ejemplo de modelo con tres vistas, en un diseño **MVC**. [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 15]

¹⁸Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*.

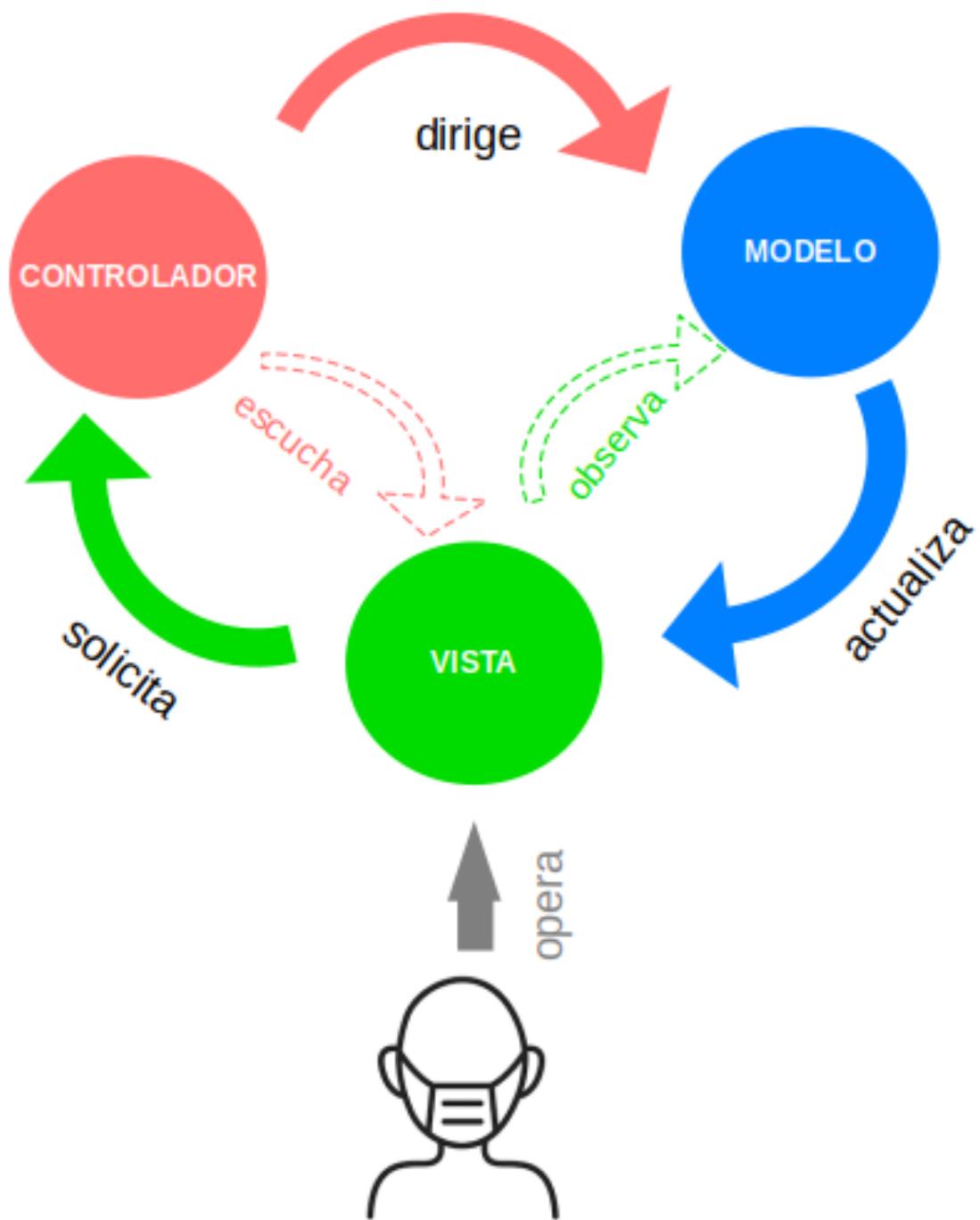


Figura 1.5: Ejemplo de diagrama relacional de la terna de clases en el diseño [MVC](#).

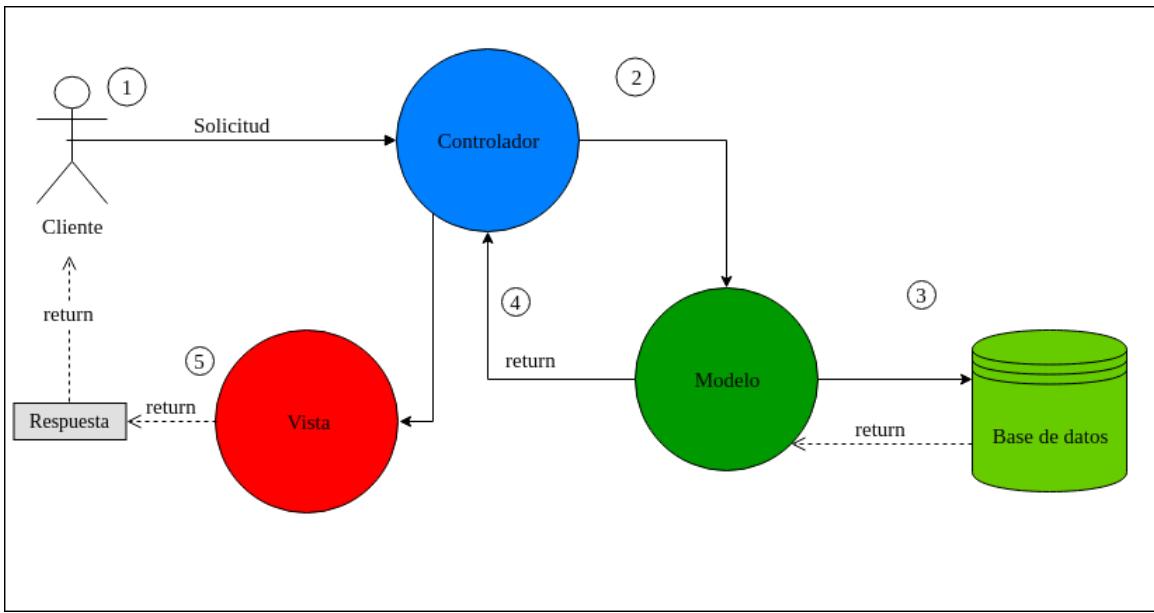


Figura 1.6: Ejemplo más complejo de diagrama relacional en el diseño **MVC**. En este caso consideramos que el sistema es multiusuario, con distintas vistas para los distintos usuarios, pero un único modelo.

- *Observer.-* Se trata de establecer un protocolo de suscripción/notificación entre el modelo y la vista para desacoplar la vista del modelo, es decir, que la vista no se asocie (no navegue hacia) el modelo. Cada vez que el modelo cambia, notifica a todas las vistas que tiene suscritas, el cambio, y las vistas se modifican a sí mismas. En la Figura 1.6 aparece un ejemplo de modelo con tres vistas.¹⁹ El patrón *observador* generaliza la idea para desacoplar cualquier tipo de objetos, y no solo los que representan la vista y el modelo en un diseño **MVC**.
- *Composite.-* Se trata de un patrón de diseño en el que un grupo de objetos son tratados como uno individual. Para ello, una clase definirá objetos simples, y otra clase definirá objetos complejos, como agregación o composición de los objetos simples o de otros compuestos. En el caso del diseño **MVC** para GUIs, una vista puede contener otras vistas (vistas anidadas), por ejemplo un panel de control de botones es una vista compuesta por vistas simples de botones. Por ejemplo, esto puede usarse para implementar un inspector de objetos (así se hace en Ruby y en Smalltalk) y reusarse en la implementación de un depurador. El patrón *compuesto* generaliza esta idea de forma que se aplica cada vez que queramos tratar un grupo de objetos de la misma forma que a sus componentes individuales, de forma que se definen clases de objetos individuales y de compuestos de objetos como herederas de una misma clase común.
- *Strategy.-* En el diseño **MVC** puede haber distintas formas de responder a las operaciones del usuario, es decir distintos algoritmos de control, representados por objetos

¹⁹Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD*.

controladores, que heredan todos de una misma clase para poder intercambiarlos incluso en tiempo de ejecución. El patrón *estrategia* generaliza esta idea de forma que se usen objetos para representar algoritmos, que hereden de una clase común, pudiendo cambiarse el algoritmo a usar en cada momento mediante el uso de otro objeto (véase Figura 1.7).



Figura 1.7: Ejemplo aún más complejo de diagrama relacional en el diseño **MVC**. En este caso consideramos que el sistema es multiusuario, con distintas vistas para los distintos usuarios, pero un único modelo, como en la Figura 1.6 pero ahora además con varias instancias del controlador.

Otros patrones de **MVC** son el Método Factoría para especificar la clase controlador y el Decorador, para, por ejemplo, añadir desplazamiento (scrolling) a una vista.

1.2. Cómo resolver problemas de diseño usando patrones de diseño

- Encontrando los objetos (clases) apropiados.- Los patrones de diseño llevan a usar clases que no forman parte del diagrama de clases de análisis porque no existen en la realidad, por ejemplo la clase Composite o la clase Strategy. La necesidad de usar de patrones no aparece en la fase de análisis ni al principio del diseño, sino en el momento en el que pensamos en diseñar un sistema flexible y reutilizable.
- Considerando distinta granularidad.- Hay objetos que representan (1) a todo un sistema o en todo caso son muy grandes y suele haber una sola instancia de ellos y (2) otros más pequeños, existiendo a veces (3) muchos objetos similares de muy poco contenido. Esto se translada a los patrones de diseño, siendo un ejemplo del primero el patrón *fachade*, uno del segundo el patrón *singleton* y uno del tercero el patrón *flyweight*.
- Especificando la interfaz de un objeto.- Muchos patrones de diseño están relacionados con la ligadura dinámica de la orientación a objetos y el hecho de que los objetos con

una misma interfaz o una parte de la misma en común, se pueden intercambiar en la parte común. Por ejemplo, si comparten la misma firma de un método (nombre del método, argumentos y valor de retorno), aunque lo implementen de forma distinta.

- Programando en función de las interfaces y no de las implementaciones. Se trata de usar las interfaces para disminuir el acoplamiento entre los sistemas (dependencias entre subsistemas). En algunos lenguajes no existe esta diferencia (como Ruby o Smalltalk), porque al no ser tipados, la interfaz de un objeto viene especificada por la clase y por tanto la herencia de interfaces es la que se define de forma implícita al definir herencia de clases. Pero en otros lenguajes, como en Java o C++, que son tipados, sí que existen diferencias entre el tipo-s (estático-s) de la variable, que definen su interfaz y las clases que las implementan, de forma que un objeto puede ser de varios tipos (estáticos), es decir, puede implementar métodos declarados en varias interfaces. En Java además las interfaces se declaran de forma explícita y por tanto también la herencia entre las mismas. En C++ la herencia entre interfaces se lleva a cabo solo mediante clases abstractas y métodos virtuales (ligadura dinámica) puros (abstractos). Algunos patrones están basados en esta diferencia entre herencia de clases y de interfaces, como el patrón *composite* y el patrón *observer*. En todo caso, siempre podemos relacionar los objetos no por la implementación de los métodos sino por la interfaz (firma) de estos métodos. Esto se consigue haciendo uso de clases abstractas (o el concepto explícito de interfaz en Java). Gracias a esto, los objetos clientes de estos objetos no tienen que saber nada sobre la implementación de los métodos a los que invocan, les basta saber que el método forma parte de la interfaz del objeto. Incluso tampoco tiene por qué conocer el tipo específico (tipo dinámico) de esos objetos. Los patrones creaciones tienen la función de permitir este desacoplamiento, con distintas propuestas para asociar una interfaz con la implementación concreta en el momento de instanciar un objeto.
- Sacando el máximo partido de los distintos mecanismos de reusabilidad de código
 - Favoreciendo la composición (diseño de caja negra) sobre la herencia (diseño de caja blanca).- Incluso cuando se cumple la relación «es un» entre dos clases, no siempre es lo mejor el uso de la herencia. Hay una tendencia a abusar de la herencia que aumenta la dependencia del código, por ejemplo los cambios del código en clases superiores a menudo obligan a cambiar el de las subclases.
 - Usando la delegación como alternativa extrema de la composición que sustituya la herencia.- En algunos casos es más indicado delegar en otros, es decir, pasar a objetos de otras clases la responsabilidad que debe tener el objeto receptor, para aumentar la reusabilidad. Algunos patrones se basan en la delegación, como el patrón *strategy* y el patrón *visitor*.
 - Usando tipos parametrizados como alternativa a la herencia.- Los genéricos o plantillas (templates) permiten un tercer modo de reutilizar código en lenguajes tipados que permiten definir clases genéricas especificando como un parámetro el tipo de los objetos que usarán de forma que en tiempo de compilación se creen

las clases ya concretas según el tipo del parámetro usado. Una aplicación de este método es por ejemplo el uso de contenedores parametrizados de modo que métodos como la ordenación de los componentes se definen de forma genérica.

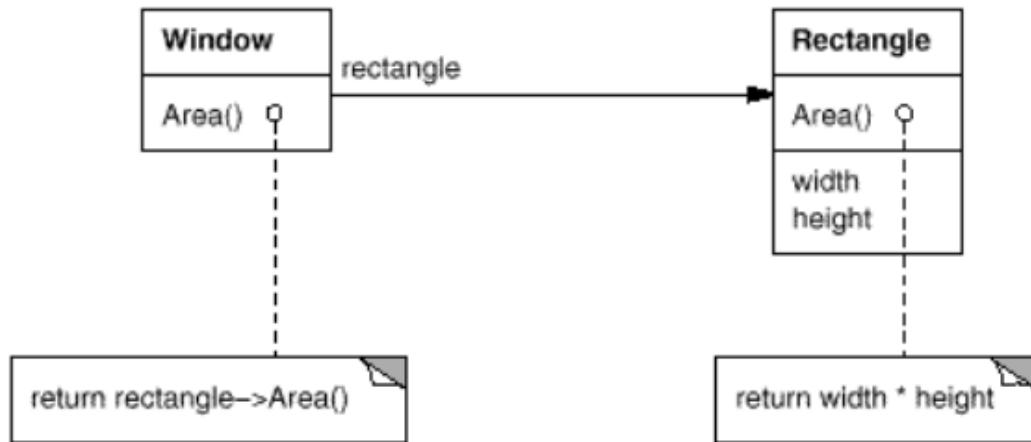


Figura 1.8: Ejemplo de uso de delegación²⁰ p. 33.

1.3. Estudio del catálogo GoF de patrones de diseño

GoF²¹ recomiendan empezar con el estudio de los siguientes patrones de diseño:

- *Abstract Factory*²² pp. 99-109 (Figura 1.12)
- *Adapter*²³ pp. 157-170
- *Composite*²⁴ pp. 183-195
- *Decorator*²⁵ pp. 196-207
- *Factory Method*²⁶ pp. 121-132
- *Observer*²⁷ pp. 326-337 (Figura 1.37)
- *Strategy*²⁸ pp. 349-359

²¹Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*.

²²Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD*.

²³Gamma et al.

²⁴Gamma et al.

²⁵Gamma et al.

²⁶Gamma et al.

²⁷Gamma et al.

²⁸Gamma et al.

- *Template Method*²⁹ pp. 360-365

Veremos además los siguientes otros patrones:

- *Prototype*³⁰ pp. 133-143 (Figura 1.19)
- *Builder*³¹ pp. 110-120
- *Visitor*³² pp. 366-381 (Figura 1.38)
- *Facade*³³ pp. 208-217

1.3.1. Patrones creacionales *Factoría Abstracta*, *Método Factoría*, *Prototipo* y *Builder*

Estos cuatro patrones son usados cuando necesitamos crear objetos dentro de un marco de trabajo o una librería software pero desconocemos la clase de estos objetos ya que depende de la aplicación concreta. GoF³⁴ también presentan un quinto patrón creacional, el patrón *Singleton*, que exige que una clase solo se pueda instanciar una vez. Sin embargo este patrón es de un ámbito mucho más reducido que los otros, afecta a una sola clase y es aplicable con cualquiera de los otros cuatro patrones creacionales. El resto de patrones creacionales están muy relacionados y es importante entender las características de cada uno para poder elegir el más adecuado a un problema concreto. Usaremos como ejemplo comparativo el juego del laberinto de GoF,³⁵ pp. 94-95 (ver Figuras 1.9 y 1.10).

²⁹Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software-* CD.

³⁰Gamma et al.

³¹Gamma et al.

³²Gamma et al.

³³Gamma et al.

³⁴Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*.

³⁵Gamma et al.



Figura 1.9: Ejemplo de un laberinto según este juego. Las puertas cerradas no son accesibles según la implementación. Es decir, al otro lado puede haber otra habitación (clase *Room*) o un muro (clase *Wall*). [Fuente: <https://www.megapixl.com/rooms-and-doors-maze-game-illustration-40888326>].

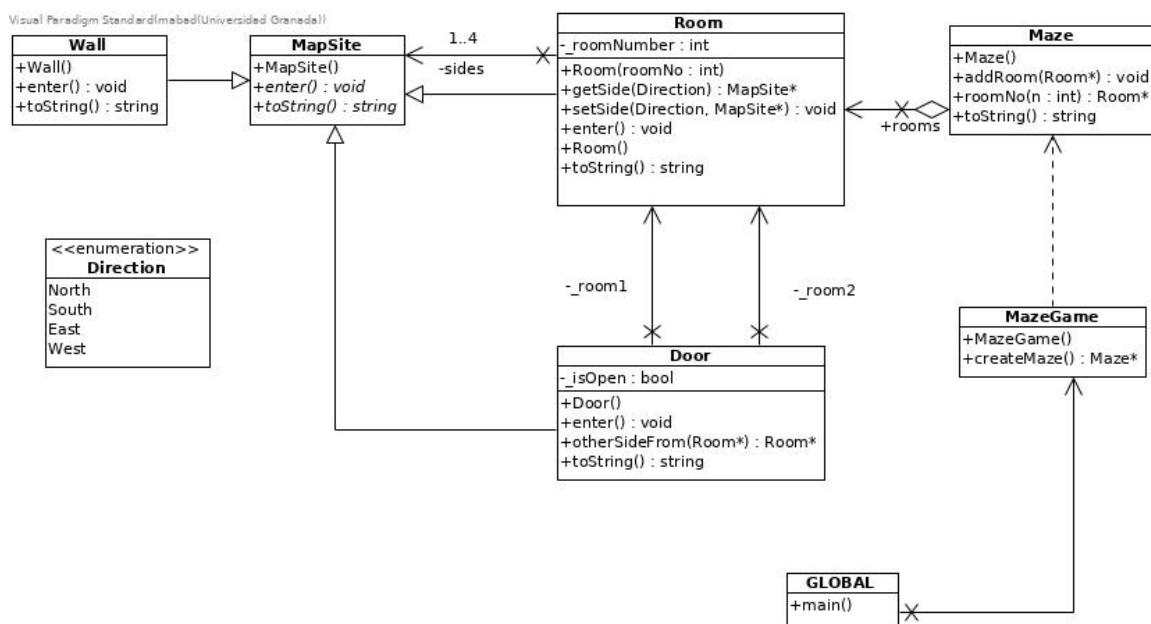


Figura 1.10: Diagrama de clases correspondiente al ejemplo del juego del laberinto de GoF.³⁶

Un ejemplo en c++ de implementación del método para crear un laberinto (*createMaze*)

muestra cómo los lenguajes **Orientado a Objetos (OO)** con métodos constructores especiales (i.e., los “falsos métodos” *new*, que no permiten polimorfismo ni ligadura dinámica) hacen un código muy poco flexible al posible uso de variantes en los objetos que se instancian (los “productos”).

```
Maze* MazeGame::createMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);
    aMaze->addRoom(r1);
    aMaze->addRoom(r2);
    r1->setSide(North, new Wall);
    r1->setSide(East, theDoor);
    r1->setSide(South, new Wall);
    r1->setSide(West, new Wall);
    r2->setSide(North, new Wall);
    r2->setSide(East, new Wall);
    r2->setSide(South, new Wall);
    r2->setSide(West, theDoor);
    return aMaze;
}
```

El concepto de factoría en **OO**

Antes de presentar estos patrones, es necesario aclarar qué significa el concepto de factoría en **OO**. El concepto puede cambiar según el tipo de lenguaje –*basado en clases* o *basado en prototipos* (este último cuando no hay clases, sino que los objetos se crean por “delegación” a partir de otros)– o el lenguaje de programación específico. Sin embargo, veremos aquí la acepción más general del concepto, que algunos consideran como un patrón de código o de bajo nivel (idiom). Una factoría es simplemente un objeto con algún método (método factoría) para crear objetos (ver Figura 1.11). En lenguajes **OO** “puros” (donde todo es un objeto), tales como Ruby o Smalltalk, esta definición contempla por tanto a la más simple de las factorías, la que crea una instancia de la propia clase, pues es un método más. Sin embargo, en los lenguajes **OO** híbridos, como Java o C++, las factorías no pueden considerarse generalizaciones de los llamados “constructores”, pues estos son métodos especiales que, además de seguir reglas sintácticas diferentes al resto de los métodos, no permiten polimorfismo ni ligadura dinámica, debiéndose explicitar la clase concreta que se quiere crear. Por tanto, el constructor de copia en Java o C++, tampoco puede considerarse un método factoría. Un ejemplo de método factoría por delegación³⁷ es el método de clonación (*clone*).

En el siguiente ejemplo se declaran los métodos factoría que clonian objetos:

```
//metodo factoria clone en clase Objeto:
Objeto clone (){
    return new Objeto(this);
```

³⁷La clase delega su responsabilidad en una instancia suya.

```

    }
// metodo factoria clone en subclase SubObjeto de Objeto:
Objeto clone (){
    return new SubObjeto(this);
}

```

Ahora puede verse que el método clone se invoca de la misma manera en una clase y su subclase:

```

// Copia de objetos con metodo factoria en una clase distinta
// 1. Creamos dos objetos de una clase y su subclase
Objeto unObjeto=new Objeto();
Objeto unSubObjeto=new SubObjeto();
// 2. Hacemos copias de ellos usando clone, un ejemplo de metodo factoria
:
Objeto otroObjeto=unObjeto.clone();
Objeto otroSubObjeto=unSubObjeto.clone();

```

Mientras que si copiamos objetos usando directamente el constructor de copia, el código no se podría compartir independientemente de la clase a instanciar, pues es específico de cada clase:

```

// Copia de objetos sin metodo factoria en una clase distinta
// 1. Creamos dos objetos de una clase y su subclase
Objeto unObjeto=new Objeto();
Objeto unSubObjeto=new SubObjeto();
// 2. Hacemos copias de ellos usando los constructores de copia:
Objeto otroObjeto=new Objeto(unObjeto);
Objeto otroSubObjeto=new SubObjeto(unSubObjeto);

```

En los lenguajes **OO** puros, las clases son objetos, y podemos usar el mismo nombre para construir un objeto de una clase como de su subclase, siendo así un método factoría pues hay ligadura dinámica y polimorfismo. Aquí podemos ver un ejemplo en Ruby con dos clases que no tienen ni siquiera que tener relación de herencia:

```

class Objeto
def self.crear
return Objeto.new
end

class Otro_objeto
def self.crear
return Otro_objeto.new
end

```

```

//El objeto o representa una clase y despues la otra. Creamos instancias
de esas clases (clases que son objetos por ser un lenguaje \acrshort{OOP})

```

```

oo} puro y tienen ligadura dinamica) usando el mismo mensaje al metodo
factoria:
o=Objeto
o.crear # crea una instancia de la clase Objeto
o=Otro_objeto
o.crear # ahora crea una instancia de la clase Otro_objeto

```

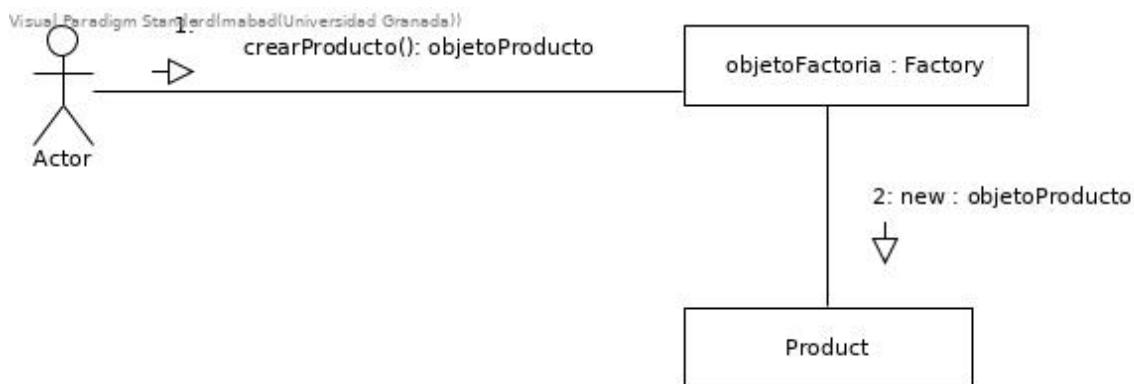


Figura 1.11: Diagrama de comunicación que muestra un ejemplo de creación de una instancia de la clase *Product* mediante el método factoría *crearProducto* de la clase *Factory*.

Los patrones creacionales tienen el objeto de permitir un código más flexible y reusable ante el posible cambio de las clases que se necesitan para crear los objetos en un programa. Por ejemplo, si quisiéramos cambiar las puertas por puertas que se abren con una palabra mágica (DoorNeedingSpelling, subclase de Door), y habitaciones por habitaciones encantadas (EnchantedRoom, subclase de Room) y quisiéramos hacer un código flexible que mantuviera el mismo método *createMaze*, podríamos usar uno de los siguientes cuatro patrones creacionales:

- *Método Factoría*.- El método *createMaze* llamará a métodos comunes ligados dinámicamente (funciones virtuales en caso de c++) para crear las habitaciones, puertas y muros necesarios, es decir, a métodos factoría que deberán crearse. Además añadiríamos una subclase de MazeGame, EnchantedMazeGame, que redefiniera esos métodos factoría (ver Figura 1.18).
- *Factoría Abstracta* con *Método Factoría*.-El método *createMaze* incluirá un parámetro (la *Factoría Abstracta*), que será usado para crear todas las habitaciones, puertas y muros necesarios, de forma que según la factoría, se crearán de un tipo u otro (ver Figura 1.15).
- *Prototipo*.- El método *createMaze* incluirá varios parámetros con los prototipos de los distintos componentes del laberinto que serán copiados para crear todas las habitaciones, puertas y muros necesarios. Si se combina con el patrón *Factoría Abstracta*, tendría un único argumento, la factoría abstracta, y se implementaría como cuando se

usa el patrón *Método Factoría*, pero esta factoría tendría prototipos (de muro, habitación y puerta) que podrían ser de distintos tipos de laberintos, creándose laberintos más variados (laberintos “mezcla”) (ver Figura 1.20).

- *Builder*.- El método *createMaze* incluirá un parámetro capaz de crear un laberinto completo usando las operaciones de añadir muros, habitaciones y puertas al laberinto, y después *createMaze* puede usar la herencia para hacer cambios en las distintas partes del laberinto o en la forma que se construye (ver Figura 1.23).

Patrón *Factoría Abstracta (kit)*

Recomendado cuando en una aplicación tenemos líneas, temáticas o familias “paralelas” de objetos que se necesitan producir (“productos”) y se prevé que puedan añadirse nuevas líneas. Con este patrón se podrá elegir una familia de entre todas las definidas sin que cambie el código al cambiar de familia elegida, y además el cliente solo tiene que conocer la interfaz de acceso a cada producto (común a todas las líneas), pero no cómo se implementa la forma de crearlos o de operar con ellos.

Sin embargo este patrón no está recomendado si se piensan agregar nuevas clases a líneas ya creadas.

Por ejemplo, una línea de clases puede ser una librería gráfica (Swing, AWT ...) o un estilo de componentes gráficos o un tipo de escritorio (GNOME o KDE en el caso de linux) y las clases pueden ser los componentes gráficos (*Boton*, *Menu*, *Panel*, ...) o elementos del escritorio (*BarraTareas*, *AreaTrabajo*, *Ventana*, *Menu*, ...). que existirán para cada librería, estilo o escritorio. El patrón utiliza una interfaz (*AbstractFactory* en la Figura 1.13) o una clase completamente abstracta (sin ningún método implementado en la propia clase). Sin embargo, también se podría usar una clase abstracta convencional, donde puede haber métodos de creación implementados en esa clase si la creación de los objetos de una clase no depende de la familia (por ejemplo, si un panel se crea de la misma forma para GNOME y para KDE). En la interfaz o clase abstracta se declara un método de creación para cada tipo de objetos (*crearProductoA*, *crearProductoB* en la Figura 1.13). En nuestro ejemplo, serían los métodos *crearBoton*, *crearMenu*, *crearPanel*. Este patrón está menos recomendado si lo que vamos a cambiar o añadir son nuevas clases (*Frame*, *Box*, ... en el ejemplo) y no líneas de clases (como un nuevo escritorio, vg. Cinnamon, en el ejemplo).

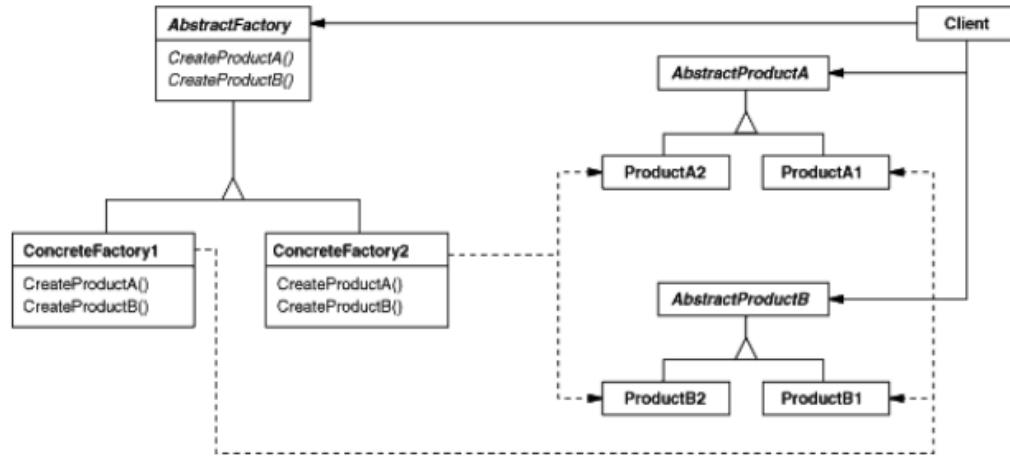


Figura 1.12: Estructura del patrón Factoría abstracta [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 101]

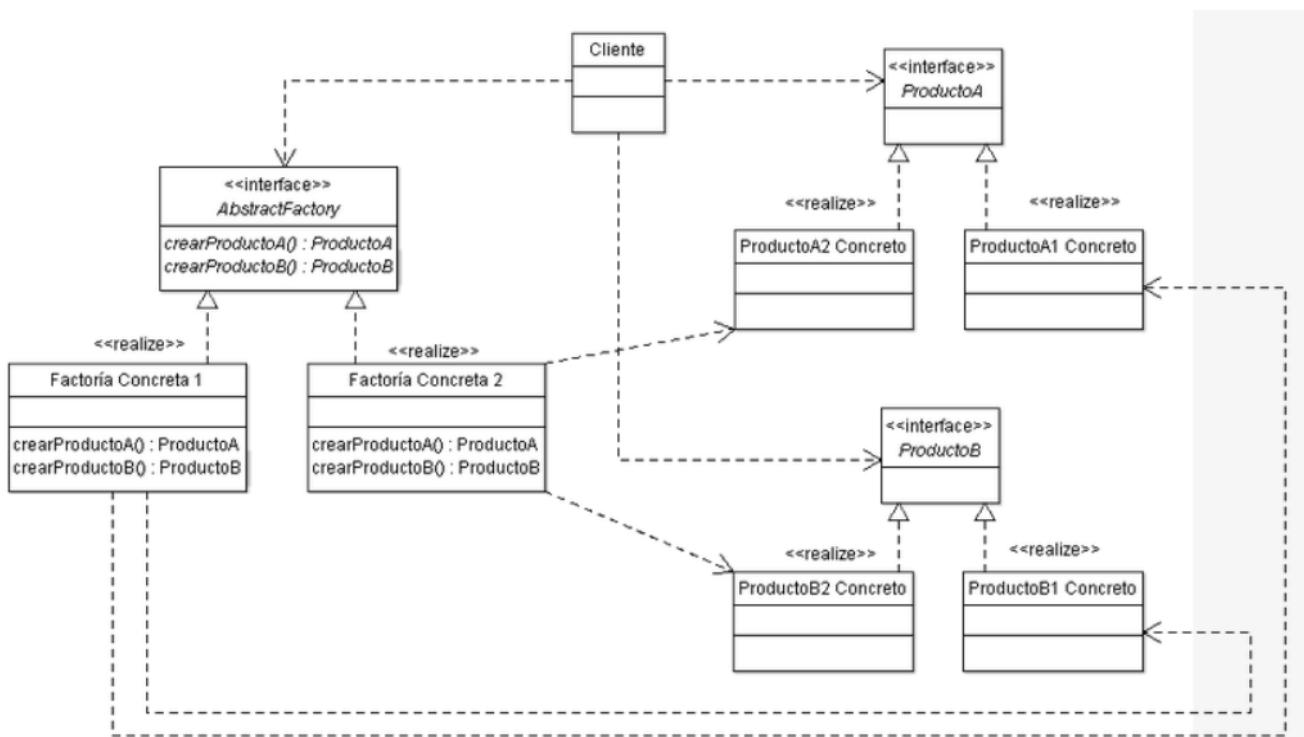


Figura 1.13: Otro diagrama de clases (más ampliado) del patrón *Factoría Abstracta*. [Fuente: [Abstract Factory](#)].

Hay dos formas en las que se pueden crear los objetos por las factorías abstractas, utilizando a su vez otros patrones creacionales: (1) patrón *método factoría*, que crea haciendo uso de métodos factoría, y (2) patrón *prototipo*, que crea siempre por clonación a partir de todas las posibles clases que se quieran poder instanciar (prototipos) (ver Figura 1.14).

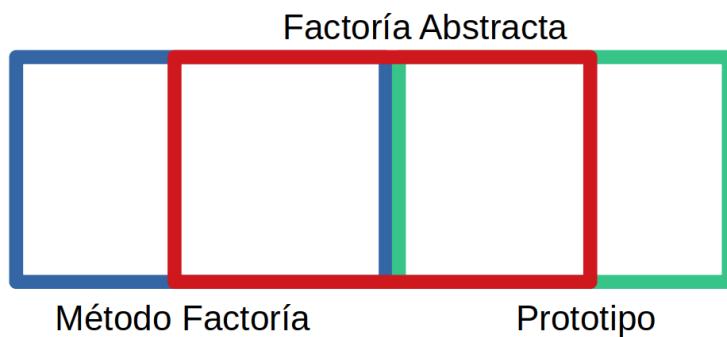


Figura 1.14: Relación de coexistencia entre los patrones *Factoría Abstracta*, *Método Factoría* y *Prototipo*.

En la Figura 1.15 puede verse el resultado de aplicar este patrón al juego del laberinto, usando métodos factoría.

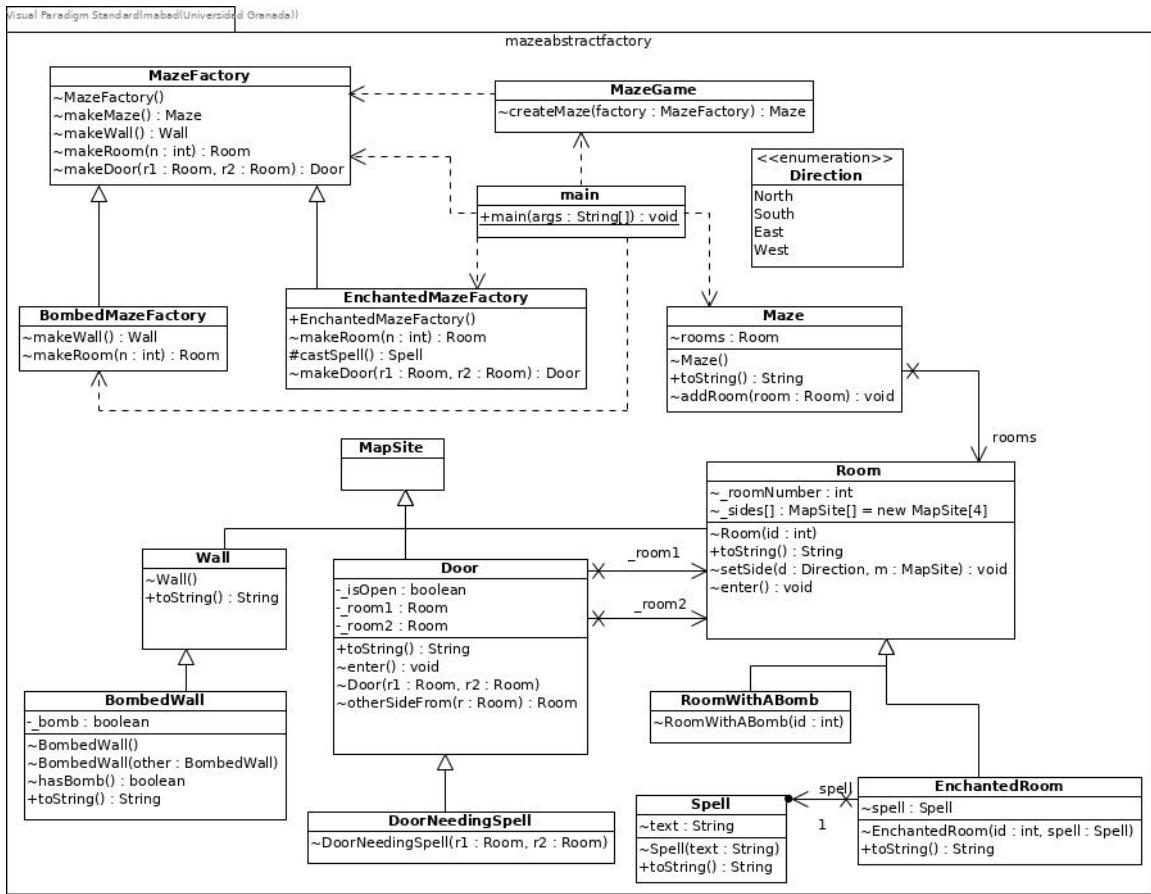


Figura 1.15: Diagrama de clases correspondiente a la aplicación del patrón *Abstract Factory* en el ejemplo del juego del laberinto de GoF.³⁸ Este patrón se usa aquí junto con el patrón *Factory Method*.

Un ejemplo en c++ (asumiendo ligadura dinámica –métodos virtuales–) de implementación del método `createMaze`, que es independiente de la factoría concreta usada, puede verse a continuación.

```

Maze* MazeGame::createMaze (MazeFactory& factory) {
    Maze* aMaze = factory.makeMaze();
    Room* r1 = factory.makeRoom(1);
    Room* r2 = factory.makeRoom(2);
    Door* aDoor = factory.makeDoor(r1, r2);
    aMaze->addRoom(r1);
    aMaze->addRoom(r2);
    r1->setSide(North, factory.makeWall());
    r1->setSide(East, aDoor);
    r1->setSide(South, factory.makeWall());
    r1->setSide(West, factory.makeWall());
    r2->setSide(North, factory.makeWall());
    r2->setSide(East, factory.makeWall());
}

```

```
r2->setSide(South, factory.makeWall());  
r2->setSide(West, aDoor);  
return aMaze;  
}
```

Patrón *Método Factoría (Virtual constructor)*

Se trata de un patrón que define métodos factoría en clases que crearán y usarán objetos de una aplicación o marco de trabajo, en vez de llamar directamente a los constructores, para que pueda beneficiarse de la ligadura dinámica y se construya el objeto en la subclase adecuada. La redefinición de los métodos factoría en distintas subclases permite crear distintas variaciones de la aplicación, según la combinación de subclases concretas elegidas para crear los objetos a partir de ellas. Este patrón es utilizado en la mayoría de las implementaciones del patrón *Factoría Abstracta*, aunque no es obligatorio (la alternativa es usar prototipos). En el patrón *Factoría Abstracta*, el patrón *Método Factoría* está implementado en la clase o interfaz *AbstractFactory* y sus subclases (ver Figuras 1.12 y 1.13).

Por otro lado, este patrón puede utilizarse sin utilizar el patrón “Factoría Abstracta”, cuando no declaramos clases factoría específicas para crear todos los objetos de una línea sino que los métodos factoría pueden agruparse con total flexibilidad.

Un caso extremo consiste en usar una única interfaz (signatura) del método factoría (una clase en la que se declara pero no se implementa), para todos los productos (ver como ejemplo la clase *AbstractCreator* en la Figura 1.17), redefiniéndose en subclases. Otra posibilidad, si hay varias clases en la aplicación sin relación de herencia entre ellas (varias clases *Producto*), es aplicar parametrización en el método factoría único para saber a qué clase debe pertenecer un objeto que se deba crear.

Un caso en el extremo contrario es permitir tantos métodos factoría como clases distintas coexistan en una instancia de la aplicación, agrupándolos todos en la misma clase (abstracta), junto con el método para crear la propia instancia de la aplicación. Esta clase es la que se llama generalmente clase “gestora” y con este patrón los métodos factoría se podrían redefinir en subclases que representen distintas variaciones de la aplicación.

Así, este patrón es útil cuando no sepamos las clases concretas de los objetos que vamos a crear, o puedan cambiar en el futuro, añadiéndose subclases, considerándose que todas heredan de una clase abstracta (clase *AbstractProducto* en la Figura 1.17) con un método de operación común a los posibles subtipos de productos (método *operacion* en Figura 1.17). Para cada nuevo producto (clases *ProductoTipo*, *ProductoOtroTipo* en la Figura 1.17) deberán crearse creadores o factorías concretas (*CreadorProductoTipo* y *CreadorOtroProducto* en la Figura 1.17).“

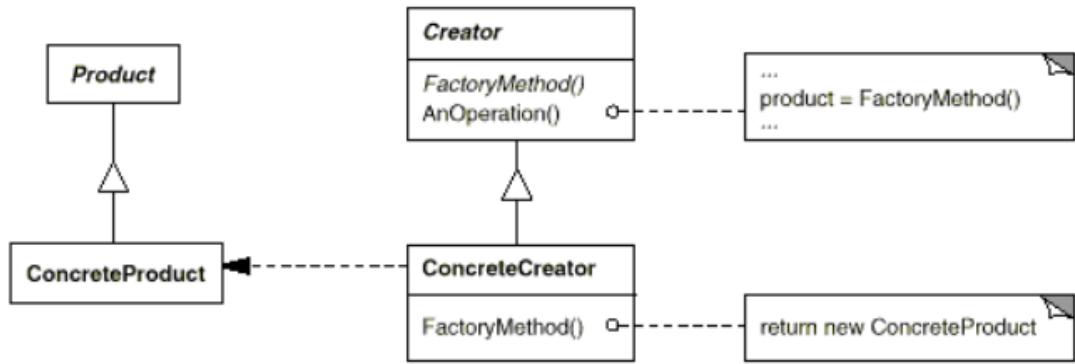


Figura 1.16: Diagrama de clases del patrón Método Factoría. [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], p. 122)].

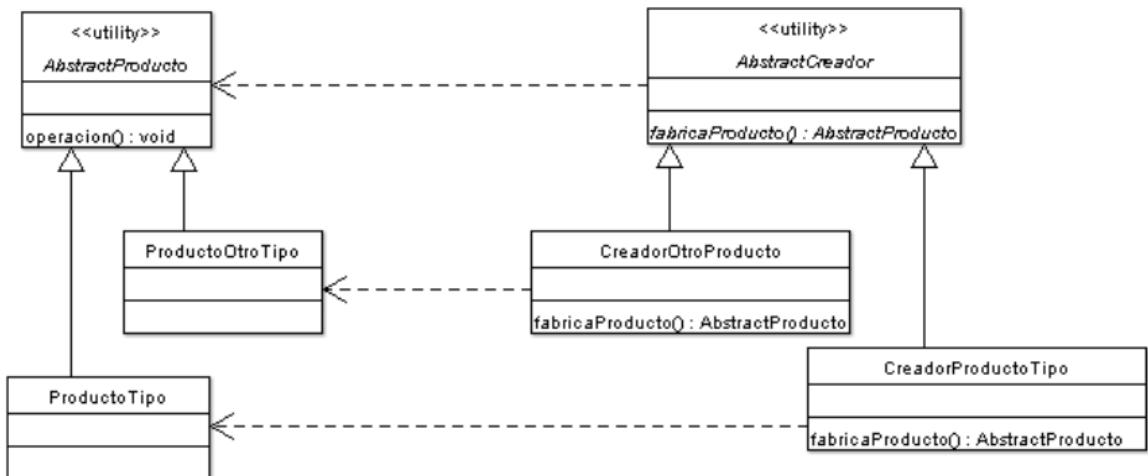


Figura 1.17: Otro diagrama de clases (más ampliado) del patrón Método Factoría. [Fuente: [Factory Method](#)].

En la Figura 1.18 puede verse el resultado de aplicar este patrón al juego del laberinto.

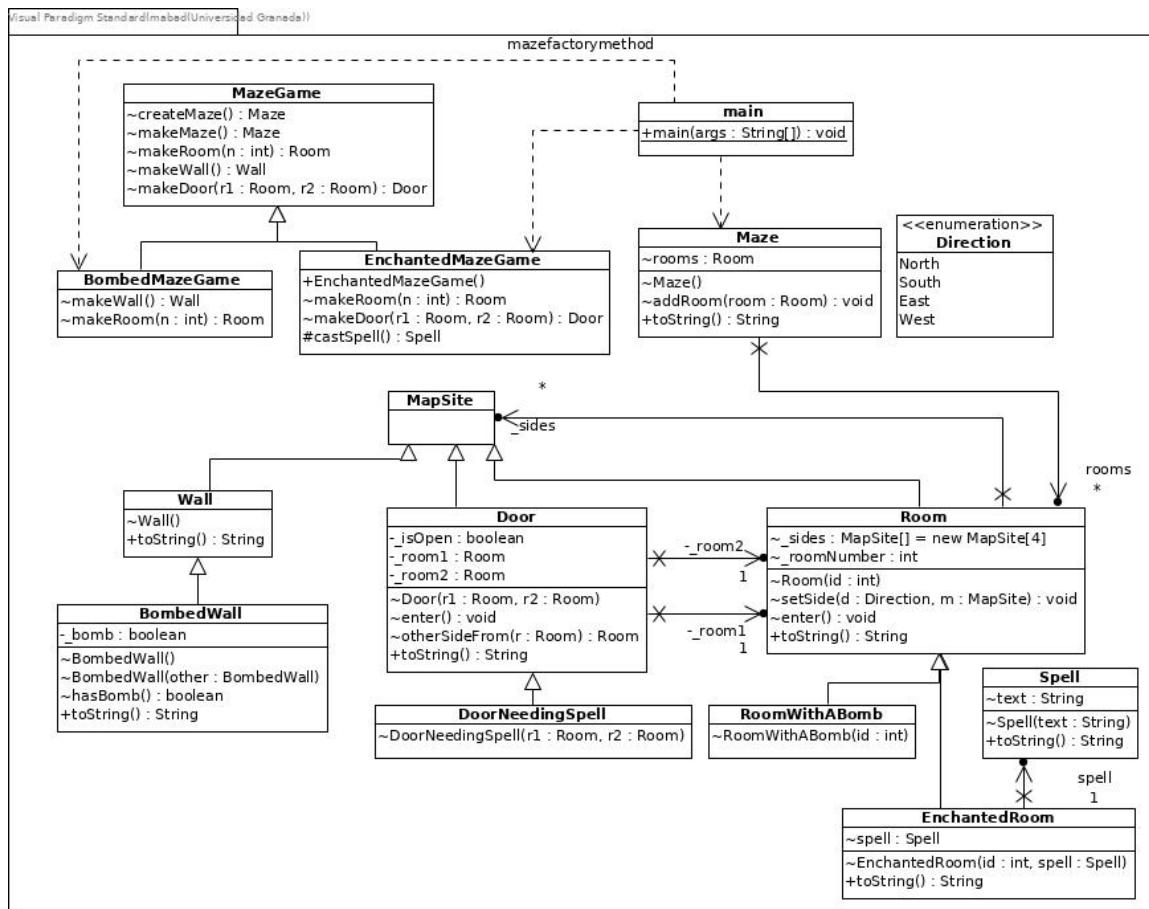


Figura 1.18: Diagrama de clases correspondiente a la aplicación del patrón *Factory Method* en el ejemplo del juego del laberinto de GoF.³⁹

Un ejemplo en c++ (asumiendo ligadura dinámica –métodos virtuales–) de implementación del método `createMaze` con el patrón *Método Factoría*, puede verse a continuación.

```

Maze* MazeGame::createMaze () {
    Maze* aMaze = makeMaze();
    Room* r1 = makeRoom(1);
    Room* r2 = makeRoom(2);
    Door* theDoor = makeDoor(r1, r2);
    aMaze->addRoom(r1);
    aMaze->addRoom(r2);
    r1->setSide(North, makeWall());
    r1->setSide(East, theDoor);
    r1->setSide(South, makeWall());
    r1->setSide(West, makeWall());
    r2->setSide(North, makeWall());
    r2->setSide(East, makeWall());
    r2->setSide(South, makeWall());
    r2->setSide(West, theDoor);
}

```

```

    return aMaze;
}

```

Para construir variantes del laberinto, por ejemplo, un laberinto encantado, se declararía una subclase de MazeGame, EnchantedMazeGame, que redefiniera los métodos para construir puertas y habitaciones, sin necesidad de redefinir el método *createMaze*.

Patrón Prototipo

Este patrón usa un prototipo para cada tipo de objeto que se vaya a usar en la aplicación, y crea uno nuevo por clonación a partir de su prototipo. Este método, considera que las clases de todos los objetos que se quieren utilizar en la aplicación heredan de la clase abstracta *Prototype*, y el cliente implementa un único método (método *operation* en la Figura 1.19) para crear objetos, pidiendo la clonación al prototipo del objeto que se desea.

Una forma alternativa al uso del patrón *Método Factoría* con el patrón *Factoría Abstracta* es la posibilidad de usar el patrón *Prototipo*. Usando prototipos no tenemos que crear la jerarquía de clases factoría concretas, paralela a la jerarquía de clases de productos concretos, sino que basta con una sola clase factoría concreta, que es la clase cliente del patrón *prototipo* con un único método de creación, que pide la clonación al prototipo del objeto que se desea crear. La única clase factoría concreta heredará de la clase factoría abstracta que deberá tener un contenedor con los prototipos de todos los tipos de objetos que en la aplicación se deseen crear.

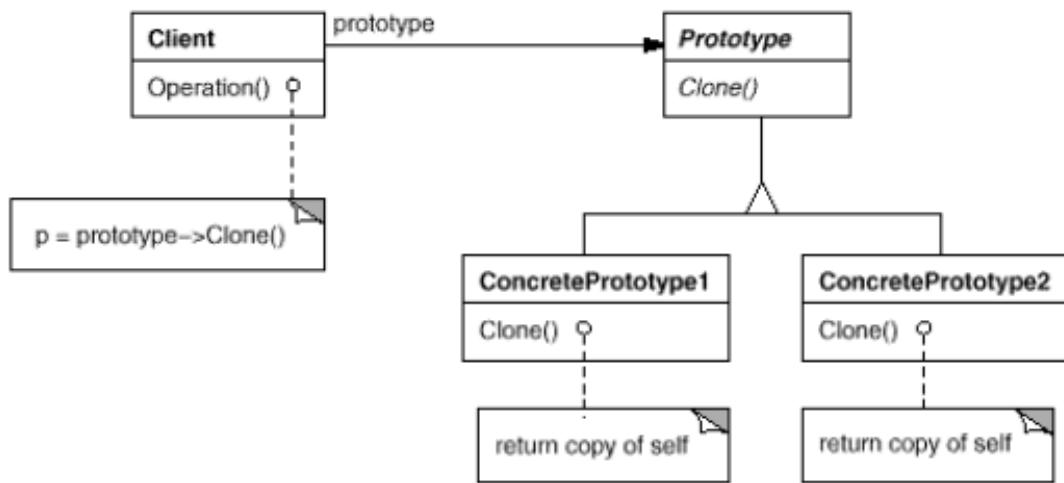


Figura 1.19: Estructura del patrón Prototype [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 135]

En la Figura 1.20 puede verse el resultado de aplicar este patrón junto con el de *Factoría abstracta* al juego del laberinto.

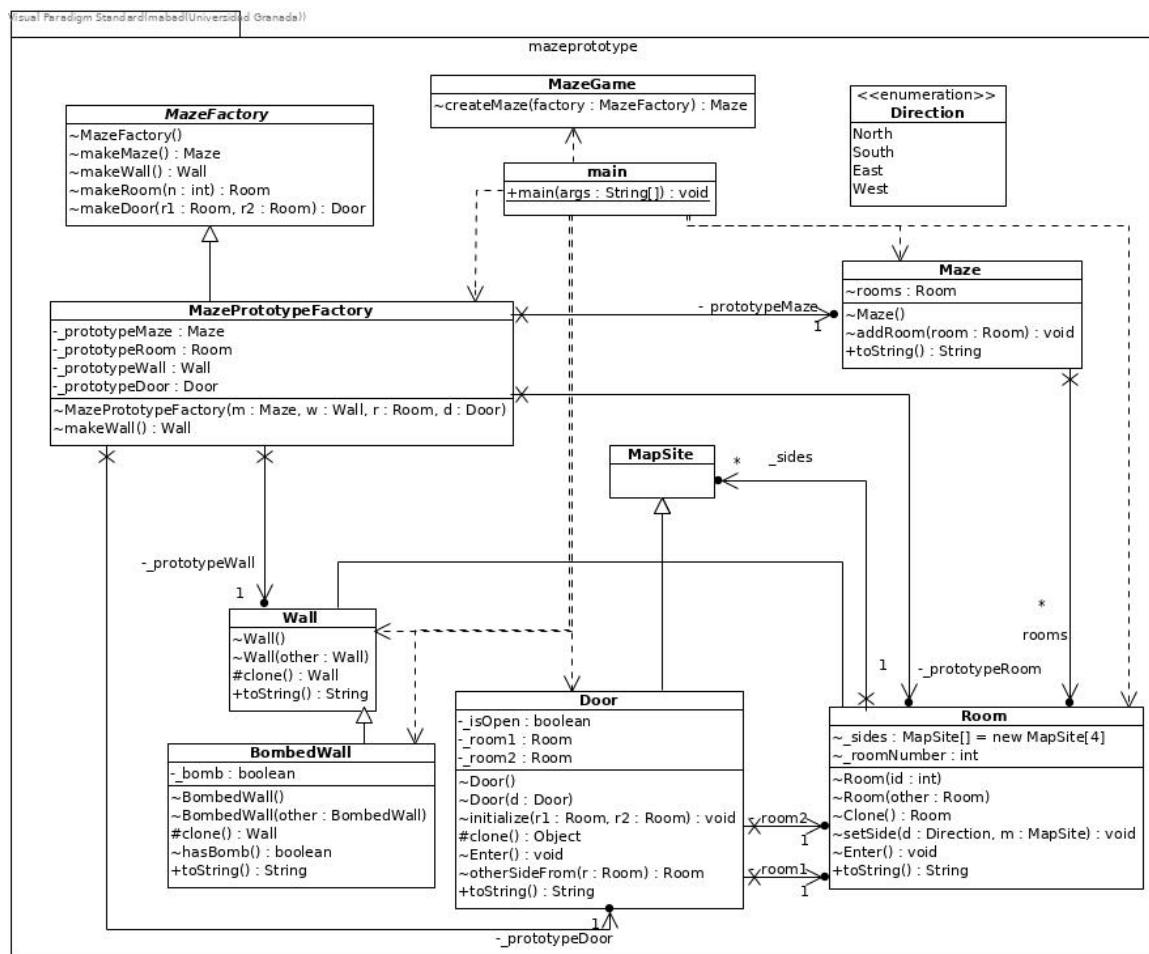


Figura 1.20: Diagrama de clases correspondiente a la aplicación del patrón *Prototype* junto con el de *Abstract Factory* en el ejemplo del juego del laberinto de GoF.⁴⁰

La implementación en c++ del método *createMaze* para el patrón *Prototype* junto con el de *Abstract Factory* sería la misma que cuando se usa la factoría abstracta con el patrón *Método Factoría*. Un ejemplo en c++ de implementación de la creación de dos factorías distintas de prototipos aparecen a continuación.

```
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);
```

```
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);
```

Patrón Builder

Este patrón se utiliza cuando queremos crear un objeto de cierta complejidad formado por componentes que puedan cambiar, así como ensamblarse y representarse de distintas formas, pero siendo común el algoritmo que se necesita para construir las partes y ensamblarlas obteniendo el objeto complejo.

Así, este algoritmo de creación es más bien lo que hace un director o guía de la construcción (un objeto de la clase Director), el cual llama a objetos constructores (builders concretos) que saben cómo construir cada una de las partes y ensamblarlas. Cada objeto concreto builder, compone y ensambla de forma distinta, pero todos comparten los métodos, por eso el director puede utilizar una interfaz común (de declaración explícita para los lenguajes tipados), que gracias a la ligadura dinámica, en tiempo de ejecución se asociará a un código concreto. La Figura 1.21 muestra el diagrama de clases con la estructura del patrón y la Figura 1.22 un diagrama de secuencias con la forma en la que cooperan el *Director* y el *Builder* con el cliente.

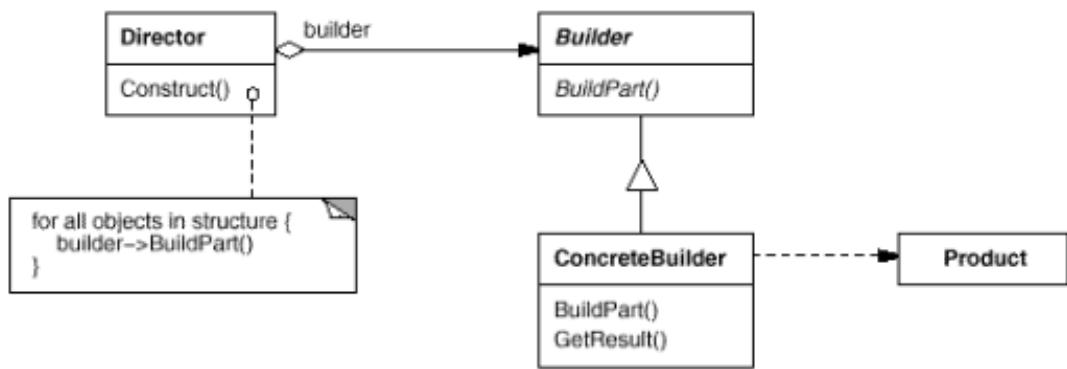


Figura 1.21: Estructura del patrón Builder [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 112]

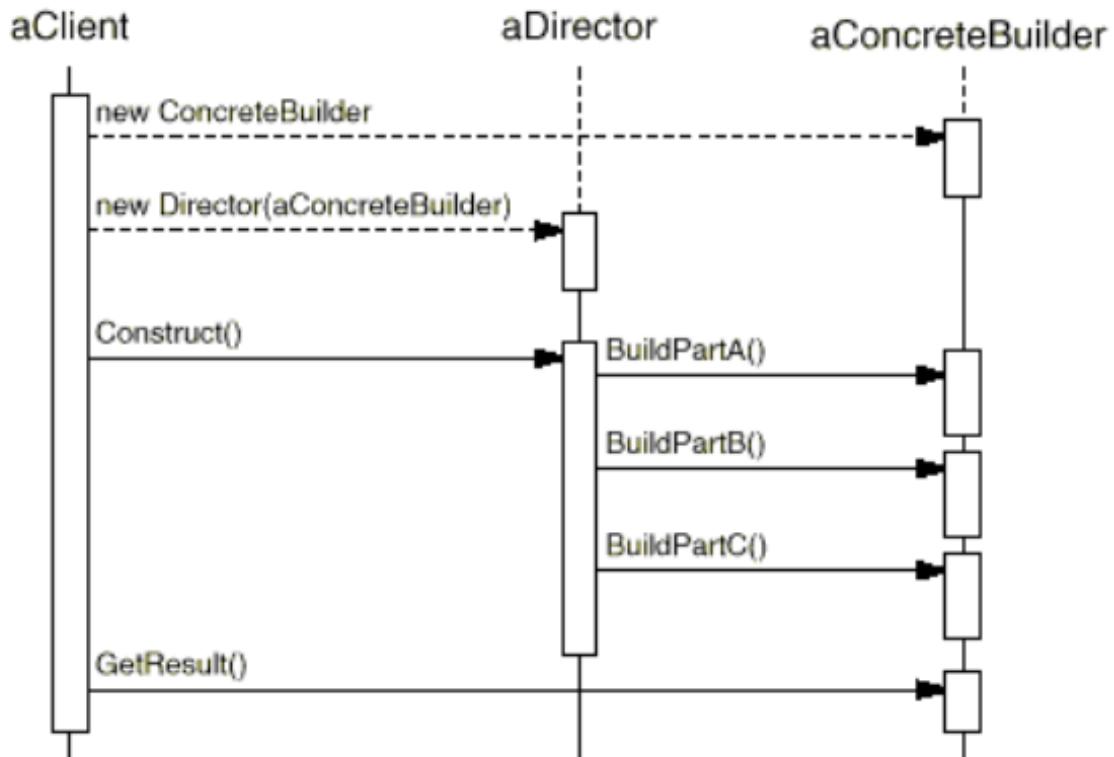


Figura 1.22: Estructura del patrón Builder [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 113]

En la Figura 1.23 puede verse el resultado de aplicar este patrón al juego del laberinto.

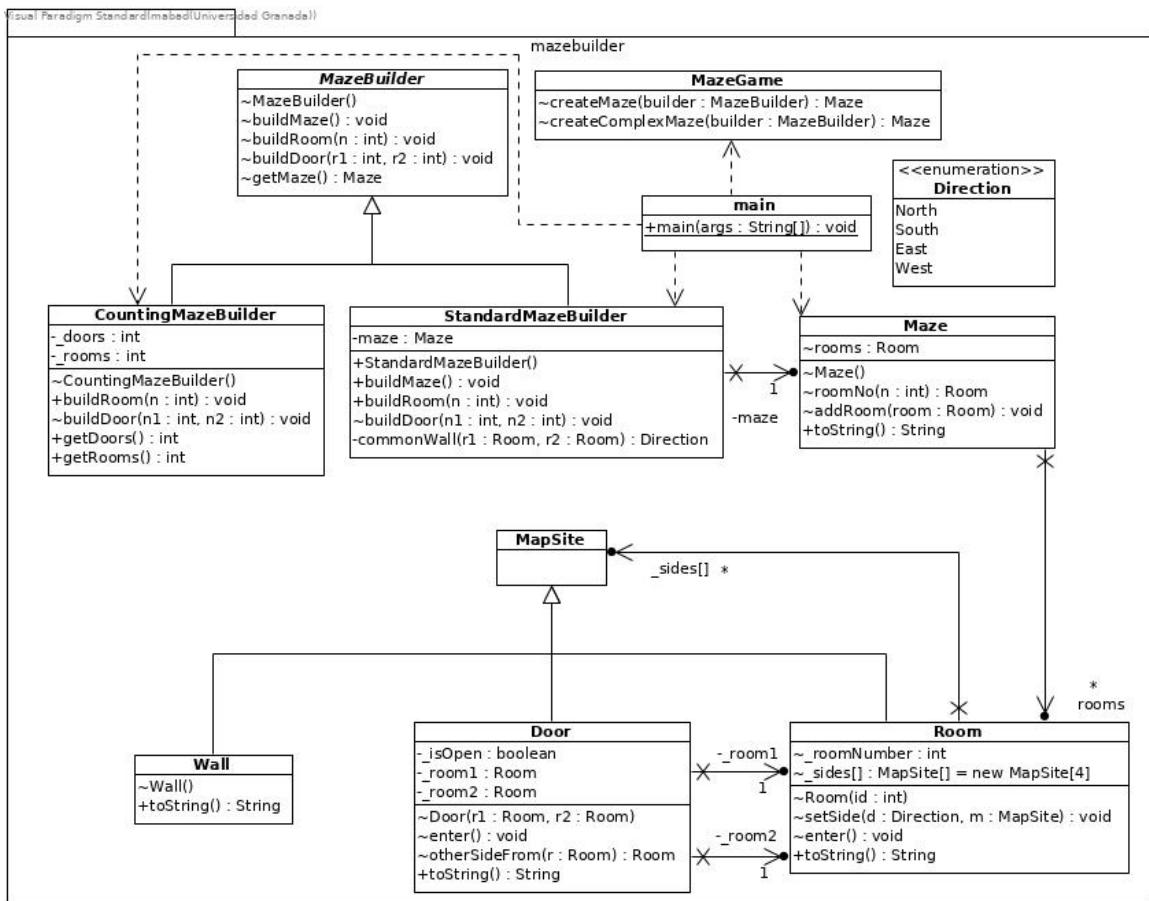


Figura 1.23: Diagrama de clases correspondiente a la aplicación del patrón “Builder” en el ejemplo del juego del laberinto de GoF.⁴¹

Un ejemplo en c++ (asumiendo ligadura dinámica –métodos virtuales–) de implementación del método `createMaze` con el patrón *Builder* puede verse a continuación.

```
Maze* MazeGame::createMaze (MazeBuilder& builder) {
    builder.buildMaze();
    builder.buildRoom(1);
    builder.buildRoom(2);
    builder.buildDoor(1, 2);
    return builder.getMaze();
}
```

Si necesitamos construir otro tipo de laberinto, por ejemplo un laberinto complejo, se puede hacer como aparece a continuación:

```
Maze* MazeGame::createComplexMaze (MazeBuilder& builder) {
    builder.buildRoom(1);
    // ...
    builder.buildRoom(1001);
    return builder.getMaze();
```

{}

CRITERIO DE CALIDAD 1.3.1: Bajo acoplamiento

El bajo acoplamiento en diseño **OO** se refiere a que haya pocas dependencias entre clases pertenecientes a subsistemas distintos.

CRITERIO DE CALIDAD 1.3.2: Alta cohesión

La alta cohesión en diseño **OO** se refiere a que existan muchas dependencias –alta conectividad– entre clases pertenecientes a un mismo subsistema.

1.3.2. Patrones estructurales *Facade*, *Composite*, *Decorator* y *Adapter*

Patrón *Fachada*

Por este patrón se proporciona una interfaz única de un subsistema o componente de un sistema, para facilitar el uso del sistema mediante el bajo acoplamiento. Es un patrón muy importante para garantizar el bajo acoplamiento. El bajo acoplamiento y la alta cohesión, son **criterios de calidad** en el diseño de software orientado a objetos. Garantizando ambos, el software se hace más reusable, mantenible y fácil de probar. En la Figura 1.24 puede verse un esquema de un diagrama de clases antes y después de la aplicación del patrón y en la Figura 1.25 la estructura de un diagrama con el patrón ya aplicado.

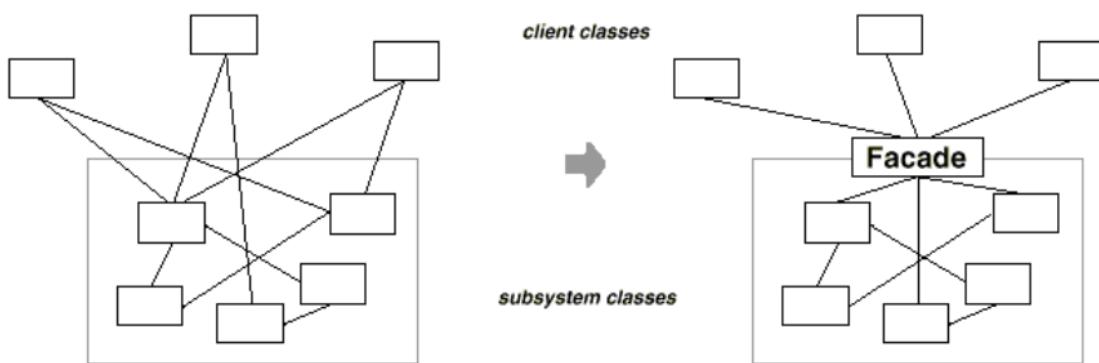


Figura 1.24: Un esquema de diagrama de clases antes y después de la aplicación del patrón Fachada [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 208].

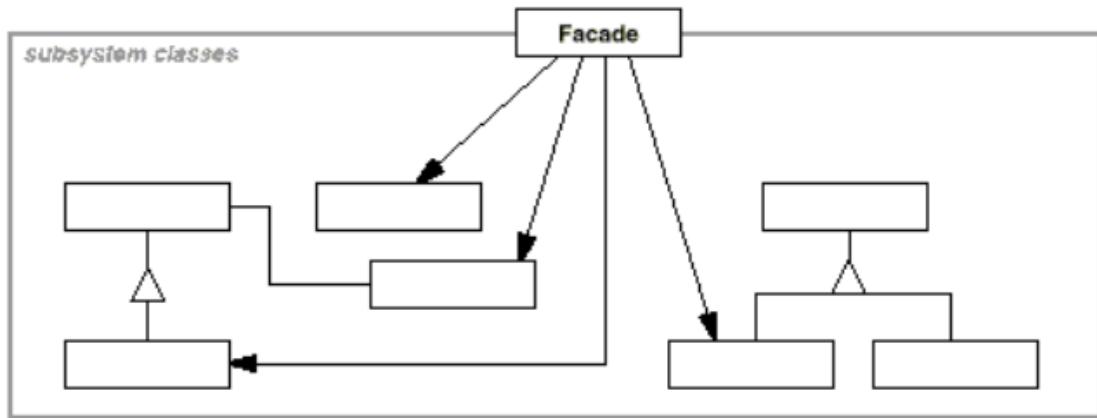


Figura 1.25: Estructura del patrón Fachada [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 210].

Patrón Composición

Permite tratar a objetos compuestos y simples de la misma forma, mediante una interfaz común a todos, que define un objeto compuesto de forma recursiva. Esto permite representar los objetos en forma de árbol. Es apropiado cuando el cliente o usuario de esos objetos no quiera distinguir si son compuestos o simples. La Figura 1.26 muestra la estructura del patrón, mientras que la Figura 1.27 muestra la estructura jerárquica de los objetos compuestos.

Un ejemplo concreto es el uso del patrón en un editor de dibujo, tal y como aparece en la Figura 1.28 con el diagrama de clases que representa la estructura del patrón y en la Figura 1.29 con una jerarquía posible de elementos gráficos según el patrón.

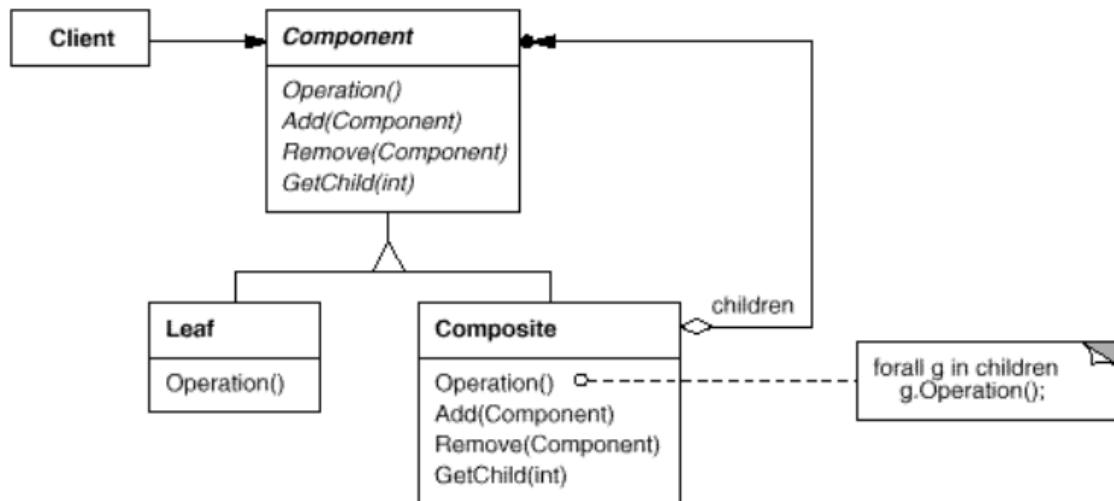


Figura 1.26: Estructura del patrón Composición [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 185].

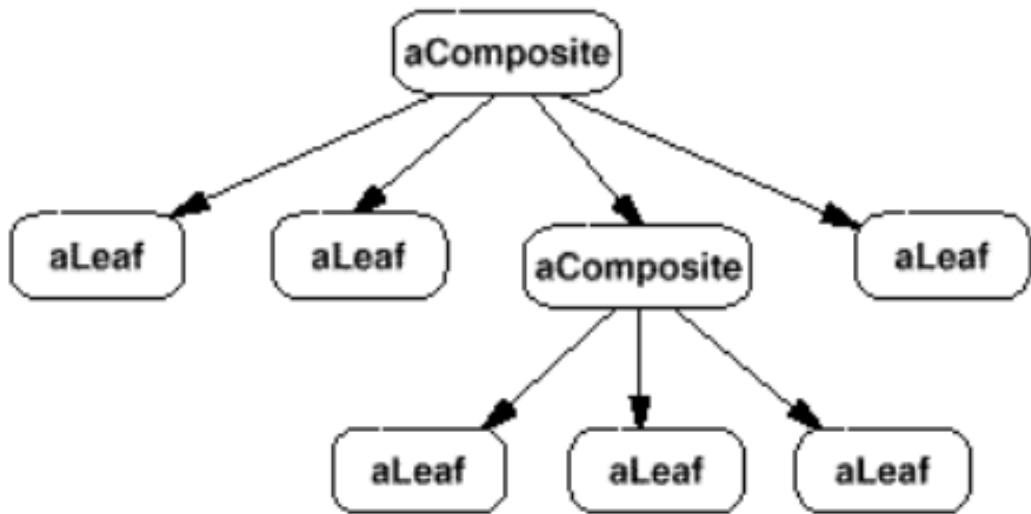


Figura 1.27: Un ejemplo de jerarquía de objetos donde se ve la relación entre objetos compuestos y simples para el patrón “Composición” [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 185].

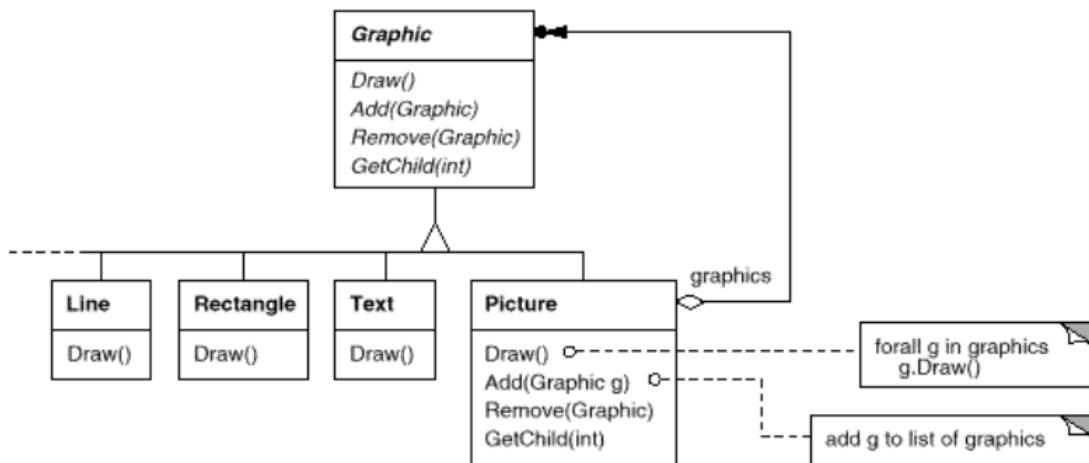


Figura 1.28: Estructura del patrón Composición en el ejemplo de componentes gráficos.
[Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], pg. 183)].

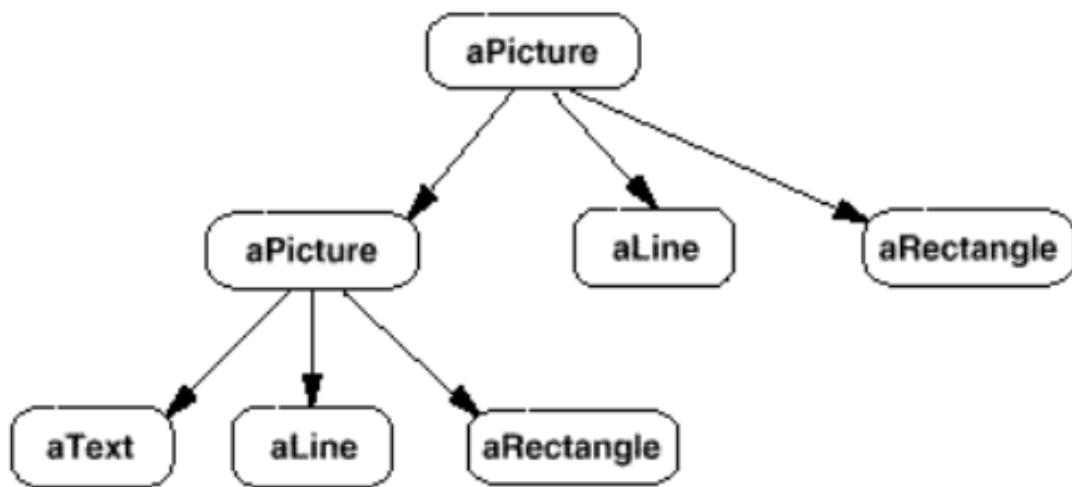


Figura 1.29: Una jerarquía de objetos en el ejemplo de componentes gráficos [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], pg. 184)].



Flutter 1.3.1. Patrón compuesto en los widgets de Flutter

Otro ejemplo de uso de este patrón es la relación entre la mayoría de los *widgets*, los distintos elementos (clases) que definen las distintas partes estáticas de la interfaz de usuario en Flutter. La Figura 1.30 muestra un ejemplo de diagrama de clases con la relación entre los distintos widgets de una aplicación Flutter.

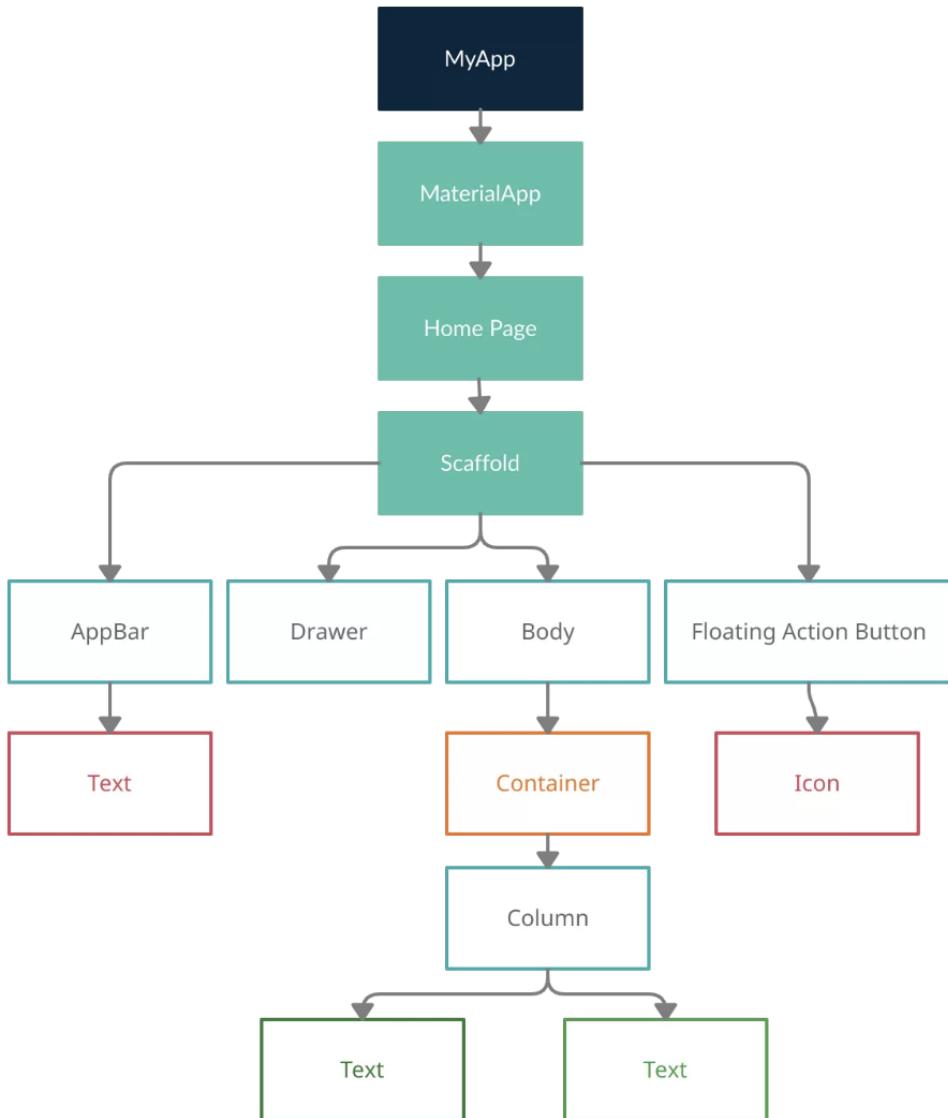


Figura 1.30: Un ejemplo de árbol de objetos widgets en una app Flutter [Fuente: <https://www.fluttercampus.com/tutorial/5/flutter-widgets/>].

Patrón Decorador (*Wrapper*)

También conocido como *Envoltorio* (del inglés, wrapper). Su función es la de proporcionar funcionalidad adicional a un objeto de forma dinámica. Esto supone una interesante alternativa a la herencia como forma de extender funcionalidad que puede añadir flexibilidad. Se aplica si queremos dotar de funcionalidad distinta a solo algunos objetos, especialmente si ésta varía, o cuando no podemos extender las clases. La alternativa si se pudieran extender las clases, podrían ser diversos criterios de subclasiﬁcación, lo que llevaría a un número demasiado elevado de subclases de las que, a su vez, podrían heredar nuevas subclases. La Figura 1.31 muestra la estructura del patrón.

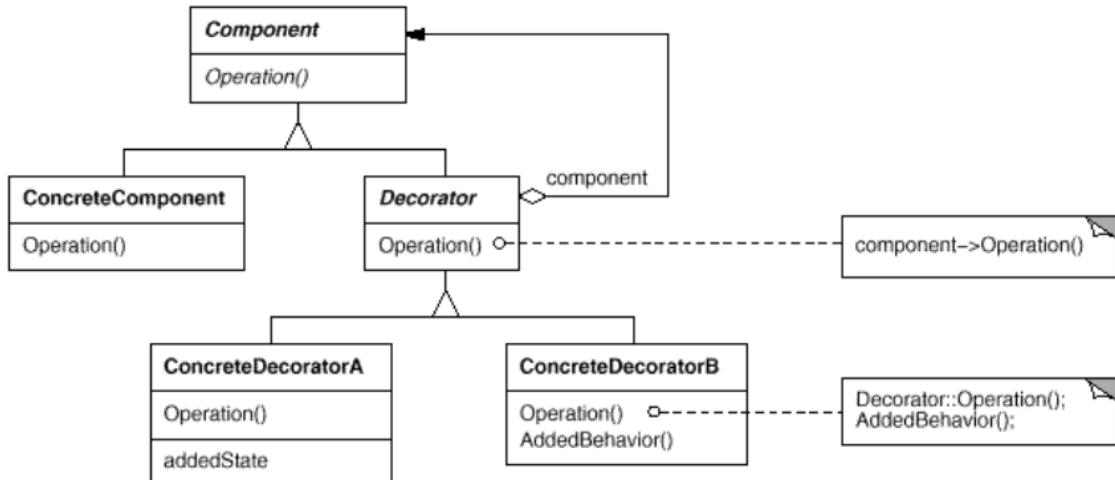


Figura 1.31: Estructura del patrón decorador [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 199].

Como se ve en la figura, un objeto de la clase *Decorator* reenvía los mensajes al objeto de la clase *Component* que está decorando. Además puede tener sus métodos propios que ejecutar antes o después de hacer el reenvío.

El mayor problema que presenta este patrón es que los sistemas creados con él son más difíciles de depurar y mantener además de ser también más difíciles de aprender a ser usados por otros.

La Figura 1.32 muestra la estructura del patrón aplicada a un ejemplo concreto de un cuadro de texto (clase *TextView*) que se “decora” con una ventana que permite hacer (clase *ScrollDecorator*) y con un borde (clase *BorderDecorator*). La aplicación y el resultado puede verse de forma gráfica en la Figura 1.33.

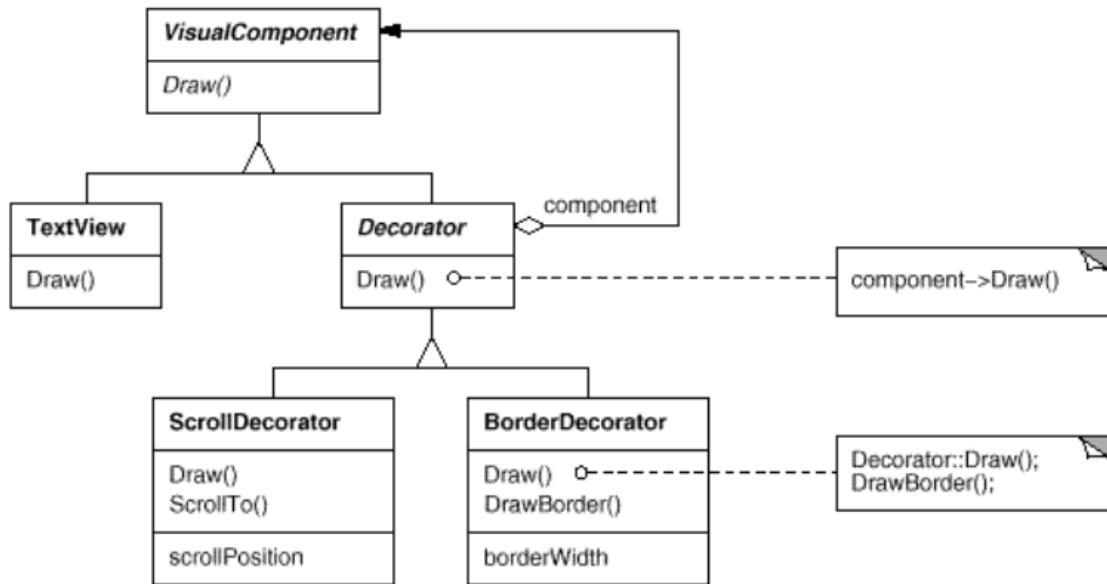


Figura 1.32: Ejemplo de estructura del patrón decorador aplicada a la presentación gráfica de un texto [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 198].

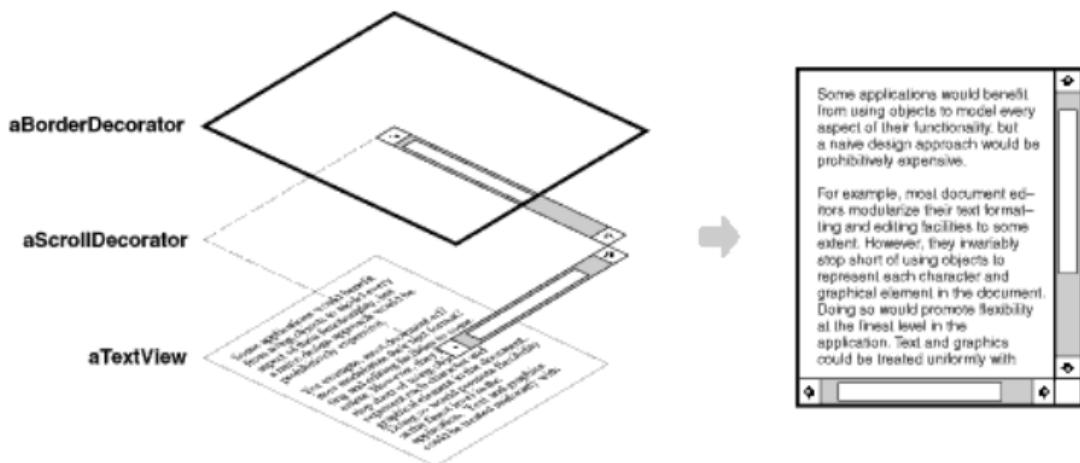


Figura 1.33: Aspecto de un cuadro de texto y sus decoradores [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], 197)].

En los fragmentos de código siguiente se puede ver una vista en Ruby on Rails que no usa el patrón decorador (Cuadro de código 1.3), que usa un decorador simple (listado 1.4) y que usa otro más complejo (Cuadro de código 1.5) para mostrar información sobre el

usuario actual. Se enfatiza en amarillo el código clave. Las clases decoradoras (en Ruby) aparecen primero (Cuadros de código 1.1 y 1.2).

```
class UserBasicDecorator < SimpleDelegator

# new methods can call methods on the parent implicitly
def info
 "#{ name } #{ lastname }"
end
end
```

Cuadro de código 1.1: Clase UserBasicDecorator

```
class UserDecorator < UserBasicDecorator
# new methods can call methods on the parent implicitly
def info
 "#{ name } #{ lastname } (created at: #{ created_at })"
end
end
```

Cuadro de código 1.2: Clase UserDecorator

```
{view/sessions/welcome.html.erb}
<h1>Clados on Rails</h1>
<% if logged_in? %>

<h3><%= "#{ current_user.name } #{ current_user.lastname }" %>
(joined: <%= current_user.created_at.strftime("%A, %d %b %Y %l:%M %p") %>)
</h3>
<%= render partial: "home_page/index" %>
<%else %>
<%= button_to "Login", '/login', method: :get %>
<%end %>
```

Cuadro de código 1.3: Sin patrón decorador

```
{view/sessions/welcome.html.erb}
<h1>Clados on Rails</h1>
<% if logged_in? %>
<% user = UserSimpleDecorator.new(current_user) %>
<h3><% user.info %></h3>
<%= render partial: "home_page/index" %>
<%else %>
<%= button_to "Login", '/login', method: :get %>
<%end %>
```

Cuadro de código 1.4: Con decorador simple

```
{view/sessions/welcome.html.erb}
<h1>Clados on Rails</h1>
<% if logged_in? %>
  user = UserDecorator.new(current_user) %>
```

```
<h3><% user.info %></h3>
<%= render partial: "home_page/index" %>
<%else %>
<%= button_to "Login", '/login', method: :get%>
<%end %>
```

Cuadro de código 1.5: Con decorador

La estructura del patrón es similar a la del patrón “Composición”, y puede parecer que este patrón es una versión degenerada del patrón “Composición” con un solo componente. Pero esto no es cierto, pues tienen funciones diferentes. El patrón “Decorador” está hecho para añadir funcionalidad a los objetos de forma dinámica mientras que el patrón “Composición” pretende unificar la interfaz de los objetos compuestos con sus componentes para que el cliente de los mismos pueda tratarlos de la misma forma.

Sin embargo, a veces pueden coexistir cuando se busque cubrir ambos objetivos, haciéndolo del siguiente modo:

«There will be an abstract class with some subclasses that are composites, some that are decorators, and some that implement the fundamental buildingblocks of the system. In this case, both composites and decorators will have a common interface. From the point of view of the Decorator pattern, a composite is a *ConcreteComponent*. From the point of view of the Composite pattern, a decorator is a *Leaf*. Of course, they don't have to be used together and, as we have seen, their intents are quite different.»⁴²

Patrón Adaptador

Este patrón convierte la interfaz de una clase en otra que se adapte a lo que el cliente esperaba para que pueda así usar esa clase.

Este patrón se suele aplicar en una fase de diseño avanzada, en la que ya hemos terminado el diseño de clases que usan otras, considerando que tienen una interfaz concreta y hemos observado que la misma funcionalidad están implementada en otro lugar, quizás en una librería o en otro módulo o paquete y queremos hacerlas reusables, de tal modo que no debamos/podamos cambiar nuestro código ni tampoco el de las clases a usar. El adaptador convertirá la interfaz de las clases externas para que puedan ser usadas por el código que ya hemos implementado.

Pero también puede aplicarse en las primeras etapas del diseño, cuando se prevé que se pueda querer usar código externo para proveer parte de la funcionalidad del software que se está desarrollando y hay razones para no usar en las clases que se desarrollan la misma interfaz de las clases que se reutilizarán, quizás porque en el futuro se reutilizarán otras con una interfaz distinta.

Hay dos versiones del patrón, de clase (ver Figura 1.34) y de objeto (ver Figura 1.35).

⁴²Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD*, pág. 247.

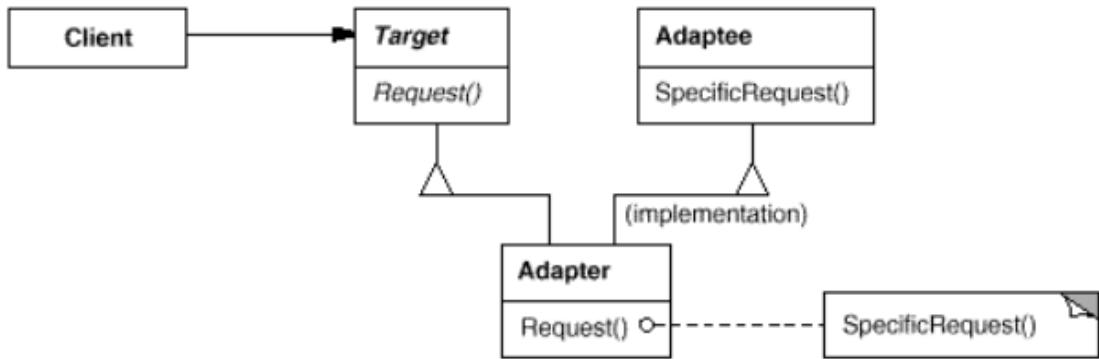


Figura 1.34: Estructura del patrón adaptador, en un ámbito de clase [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], pg. 159)].

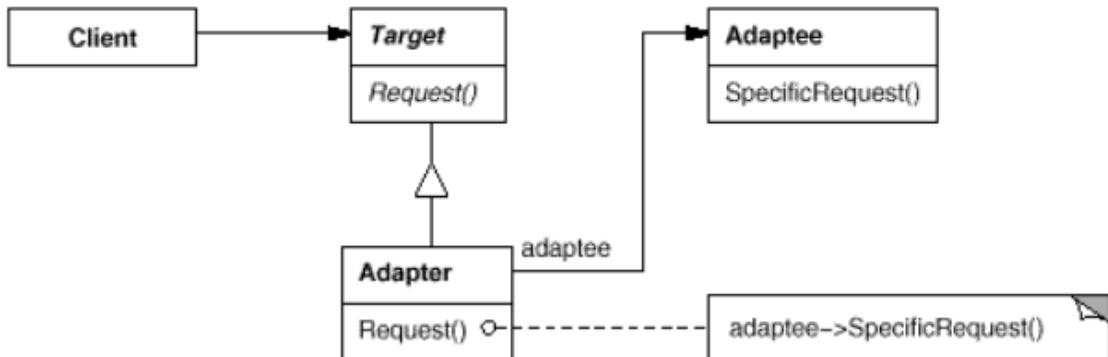


Figura 1.35: Estructura del patrón adaptador en un ámbito de objeto [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], pg. 159)].

La versión del ámbito de objeto está más indicada si queremos usar una gran variedad de subclases ya existentes. Con este patrón, se proporcionará una interfaz única adaptando la interfaz de la clase padre. La versión del ámbito de clase requiere del uso de herencia múltiple.

En la Figura 1.36 se muestra la estructura del patrón “Adapter” aplicado a un editor de dibujo. El software externo tiene un componente de texto pero con un nombre de clase y métodos distintos a los esperados.

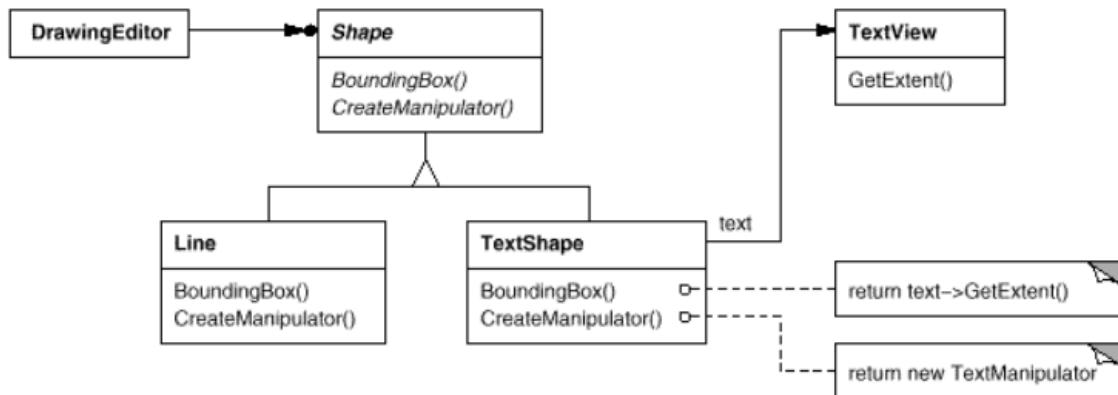


Figura 1.36: Estructura del patrón “Adapter” aplicado a un editor de dibujo [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], pg. 158)].

1.3.3. Patrones conductuales *Observer*, *Visitor*, *Strategy*, *TemplateMethod* e *InterceptingFilter*

Patrón *Observador*

Este patrón también se conoce con el nombre de *Dependientes* o *Publicar – Subscribir*. La idea es que, cuando un objeto tiene varios objetos que dependen de él, la consistencia entre ellos se garantice sin que sea a costa de aumentar el acoplamiento. Así, cada cambio en el estado del primer objeto –llamado también “sujeto observable”– es notificado, no a cada uno de ellos directamente, sino a modo de publicación común a todos los objetos en una lista de suscriptores (todos los objetos dependientes, los observadores), de forma que todos ellos recibirán simultáneamente la notificación por estar suscritos. Se pueden subscribir cualquier número de observadores a la lista de suscripción del sujeto observable.

En este patrón se define una interfaz para los observables que declara métodos para añadir o quitar suscriptores y para notificarles los cambios (clase abstracta *Subject* en la Figura 1.37) y otra interfaz para los observadores que declara un método de actualización cada vez que se les notifica un cambio (clase abstracta *Observer* en la Figura 1.37).

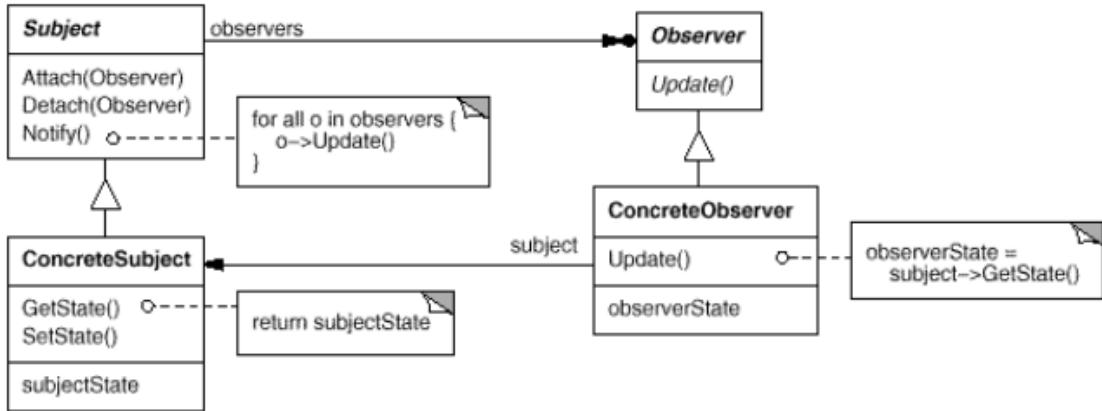


Figura 1.37: Estructura del patrón Observador [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994]), p. 328]

Este patrón se suele aplicar para desacoplar la funcionalidad de un sistema de su presentación al usuario. De este modo, varias interfaces distintas de usuario se pueden implementar sin tener que tocar la funcionalidad de la aplicación. Es un patrón que se aplica a menudo junto con el diseño arquitectónico [MVC](#), que es considerado también un patrón, pero a nivel arquitectónico.

Patrón Visitante

Este patrón busca separar un algoritmo de la estructura de un objeto. La operación se implementa de forma que no se modifique el código de las clases que forman la estructura del objeto. Este patrón debe utilizarse cuando se quieren llevar a cabo muchas operaciones dispares sobre objetos de una estructura de objetos sin tener que incluir dichas operaciones en las clases. Dado que este patrón separa un algoritmo de la estructura de un objeto, es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general. Se debe utilizar este patrón si se quiere realizar un cierto número de operaciones, que no están relacionadas entre sí, sobre instancias de un conjunto de clases, y no se quiere “contaminar” a dichas clases.

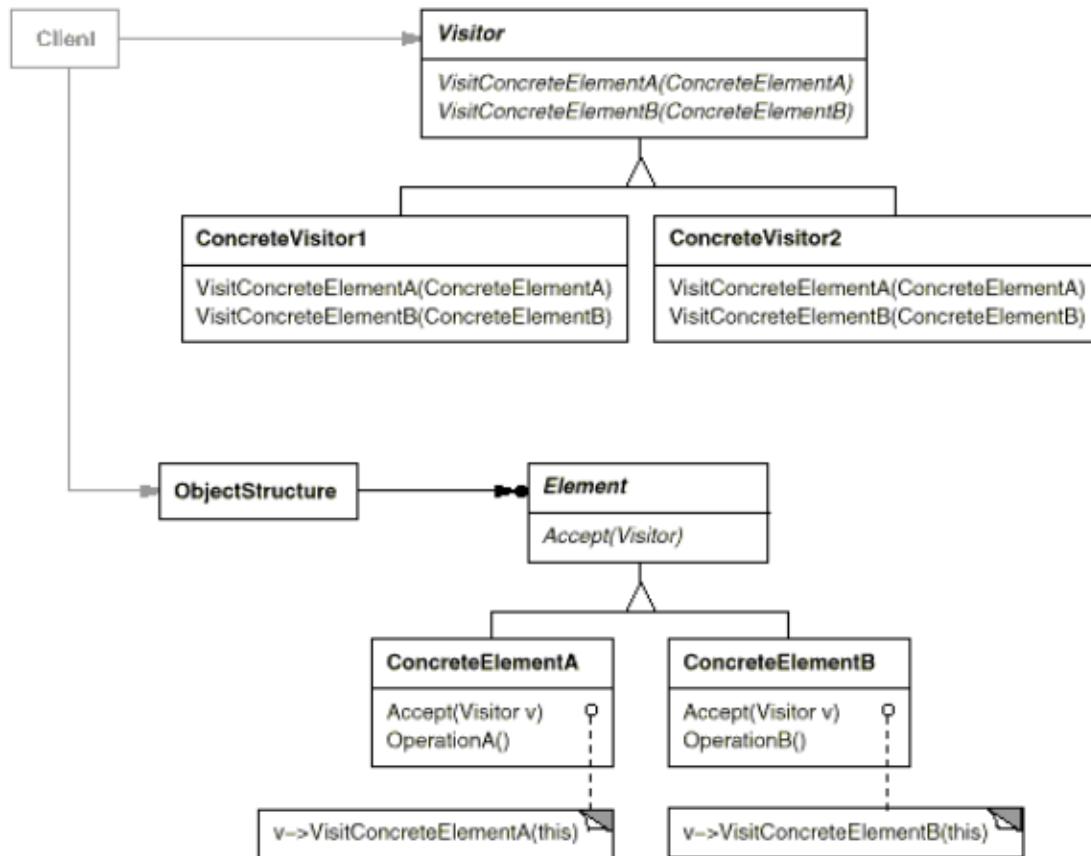


Figura 1.38: Estructura del patrón “Visitante” [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], pg. 369)]

Patrón Estrategia

Este patrón define una familia de algoritmos, con la misma interfaz y objetivo, de forma que el cliente puede intercambiar el algoritmo que usa en cada momento.

Se aplica cuando se prevé que la lista de algoritmos alternativos pueda ampliarse, y/o cuando son muchos y no siempre se usan todos. Así, en vez de que formen parte de una clase Cliente, se proporcionan como clases aparte accesibles por parte del Cliente a través de una interfaz común a todos. La Figura 1.39 muestra la estructura del patrón. La Figura 1.40 muestra la estructura cuando se ha aplicado a un ejemplo de algoritmos para partir el texto por líneas en un programa que muestra el aspecto final del texto (visor de textos). La clase *Composition* tiene la responsabilidad de mantener y actualizar los saltos de línea que se muestran en el visor.

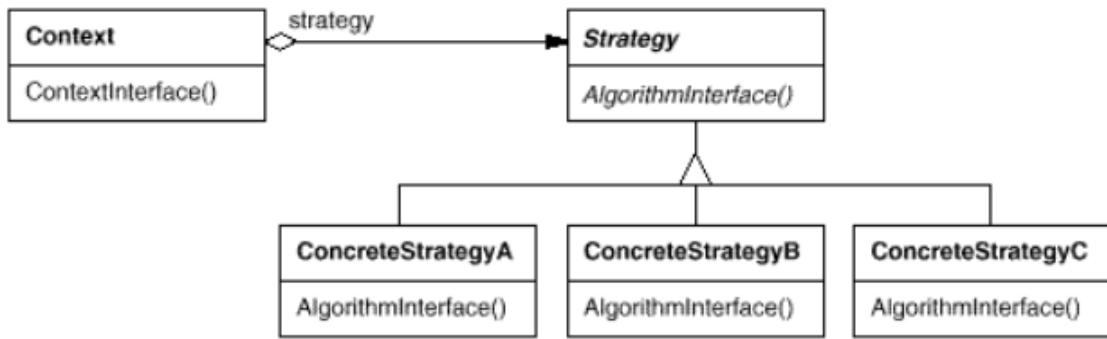


Figura 1.39: Estructura del patrón “Estrategia” [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], pg. 351)]

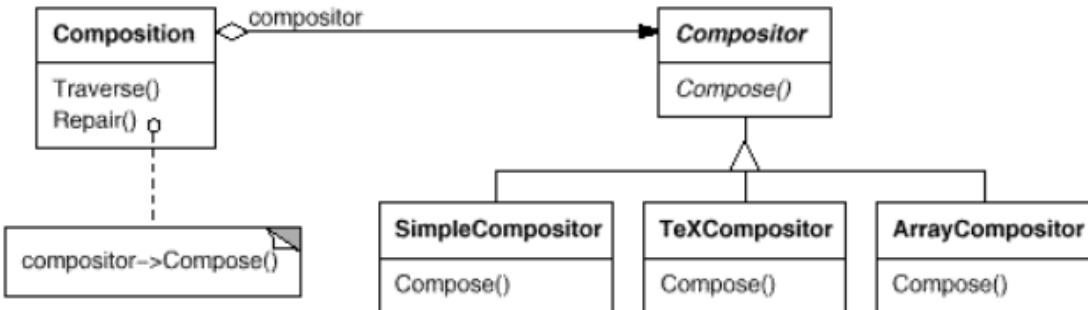


Figura 1.40: Estructura del patrón “Estrategia” en un ejemplo de visualización de textos [Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], pg. 349)]

Patrón Método Plantilla

Un método plantilla es un método de una clase abstracta⁴³ que define una secuencia de pasos (métodos) que son usados por un algoritmo de la clase, y deja la implementación de cada uno de los métodos concretos a las subclases.

Es una técnica básica muy importante de reutilización de código. Permite además poder controlar qué métodos que forman parte del algoritmo se podrán sobreescribir en las subclases –los llamados métodos gancho (del inglés hook)– y cuáles deben sobreescribirse forzosamente, declarando estos últimos como abstractos en la clase.

La Figura 1.41 muestra un diagrama de clases con la estructura de este patrón. Los métodos plantilla pueden llamar a distintos tipos de operaciones⁴⁴:

- Métodos concretos de las subclases *ConcreteClass*

⁴³Forzosamente será parcialmente abstracta pues la clase debe al menos implementar el método plantilla.

⁴⁴Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD*, pg. 363.

- Métodos implementados en la clase *AbstractClass*
- Métodos abstractos (declarados pero no implementados) en la clase *AbstractClass*
- Métodos factoría
- Métodos gancho: aquéllos implementados en la clase abstracta (aunque a menudo no hacen nada) que pueden ser sobreescritos en las subclases

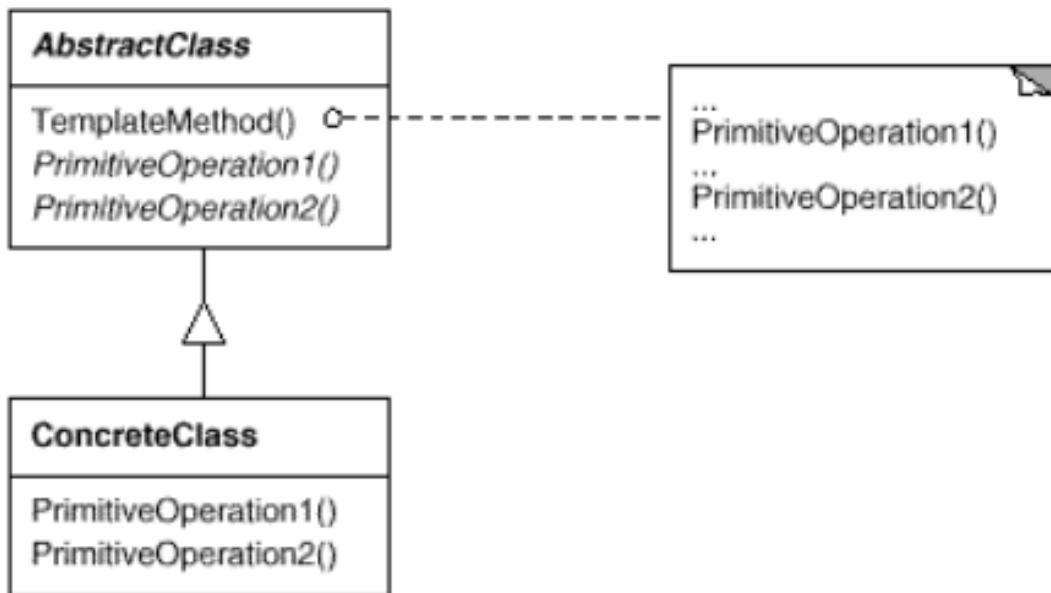


Figura 1.41: Estructura del patrón “Método Plantilla”[Fuente: (Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software- CD* [USA: Addison-Wesley Longman Publishing Co., Inc., 1994], pg. 362)]

El uso de métodos gancho evita que los métodos de las subclases que extienden a los de la clase olviden llamar al método que extienden.⁴⁵ Una extensión normal de un método *operation* de una *ParentClass* en una *DerivedClass* se implementaría de la siguiente forma en C++:

```

void DerivedClass::operation () {
// DerivedClass extended behavior
// ...
ParentClass::operation();
}
  
```

Usando un método gancho *hookOperation* que se llama desde el método *operation* de la clase padre, no hay que llamar al método *operation* desde la subclase:

⁴⁵Gamma et al., pg. 363.

```
void ParentClass::operation () {
// ParentClass behavior
hookOperation();
}
```

hookOperation no hace nada en la clase padre:

```
void ParentClass::hookOperation () { }
```

Las subclases lo extienden:

```
void DerivedClass::hookOperation () {
// derived class extension
// ...
}
```

Patrón *Filtros de intercepción*

Algunos patrones surgen como combinación de otros. Este patrón surge por la combinación de otros dos, que, a su vez, se pueden considerar tanto patrones de diseño como estilos arquitectónicos. El patrón *Filtros de intercepción* (ver Figura 1.42) está basado en (1) el patrón Interceptor, que permite añadir servicios de forma transparente que puedan ser iniciados de forma automática y en (2) el patrón arquitectónico *Tubería y Filtro*, que encadena los servicios de forma que la salida de uno es el argumento de entrada del siguiente. En concreto, el patrón Filtros de intercepcion permite añadir un componente de filtrado antes de la petición de un servicio (pre-procesamiento) o después su respuesta (post-procesamiento). Aunque los filtros generalmente implican que la petición del servicio se cancele si no se pasa el filtro, a veces pueden simplemente añadir funcionalidad dejando siempre pasar al servicio principal⁴⁶. Los filtros se programan y se utilizan cuando se produce la petición y antes de pasar tal petición a la aplicación “objetivo” que tiene que procesarla. Por ejemplo, los filtros pueden realizar la autenticación/autorización/conexión(login) o trazar la petición antes de pasarla a los objetos gestores que van a procesarla.

Como puede verse en la Figura 1.42, este patrón tiene un *GestorDeFiltros* (*FilterManager*), una *CadenaDeFiltros* (*FilterChain*) y varios objetos *Filtro*, que son los componentes de procesamiento que interceptan la petición de la tarea principal de la clase Objetivo (*Target*) que solicita la clase *Cliente*. La *CadenaDeFiltros* proporciona varios filtros al *GestorDeFiltros* y los ejecuta en el orden en que fueron introducidos en la aplicación. El *GestorDeFiltros* se encarga de gestionar los filtros (crea la cadena de filtros y tiene métodos para añadir filtros concretos y que se ejecute la petición por los filtros y el objetivo).

⁴⁶Cf. <https://www.oracle.com/technetwork/java/interceptingfilter-142169.html>

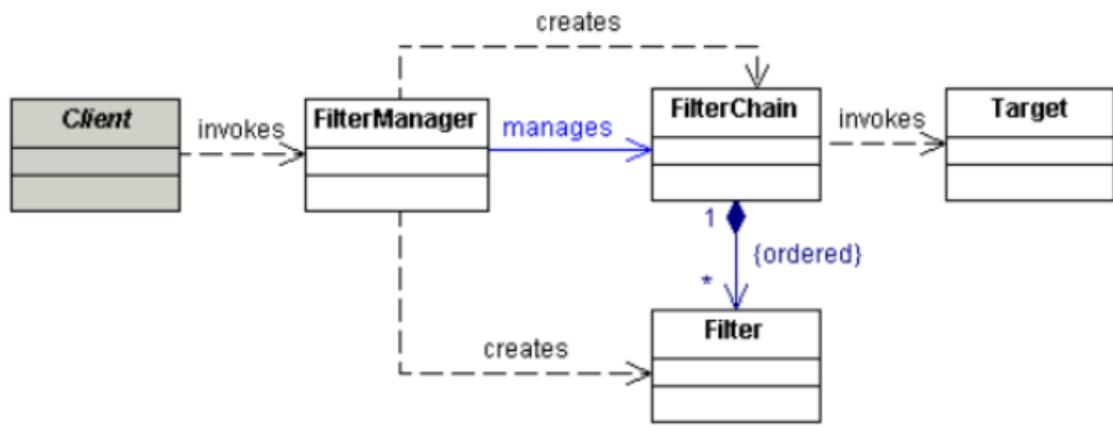


Figura 1.42: Diagrama de clases correspondiente al patrón interceptor de filtros. [Fuente: [Patrón Filtros de intercepción.](#)]

A continuación se explican las entidades de modelado necesarias para programar el patrón *Filtros de intercepción*.

- *Objetivo (target)*: Es el objeto que será interceptado por los filtros.
- *Filtro*: Interfaz (clase abstracta) que declara el método *ejecutar* que todo filtro deberá implementar. Los filtros que implementan la interfaz se aplicarán antes de que el objetivo (objeto de la clase *Objetivo*) ejecute sus tareas propias (método *ejecutar*).
- *Cliente*: Es el objeto que envía la petición a la instancia de *Objetivo*, pero no directamente, sino a través de un gestor de filtros (*GestorFiltros*) que envía a su vez la petición a un objeto de la clase *CadenaFiltros*.
- *CadenaFiltros*: Tiene una lista con los filtros a aplicar, ejecutándose en el orden en que son introducidos en la aplicación. Tras ejecutar esos filtros, se ejecuta la tarea propia del objetivo (método *ejecutar* de la clase *Objetivo*), todo dentro del método *ejecutar* de *CadenaFiltros*.
- *GestorFiltros*: Se encarga de gestionar los filtros: crea la cadena de filtros y tiene métodos para añadir filtros concretos y que se ejecute la petición por los filtros y el “objetivo” (método *peticionFiltros*).

La Figura 1.43 muestra un ejemplo de diagrama de secuencias donde se usa este patrón.

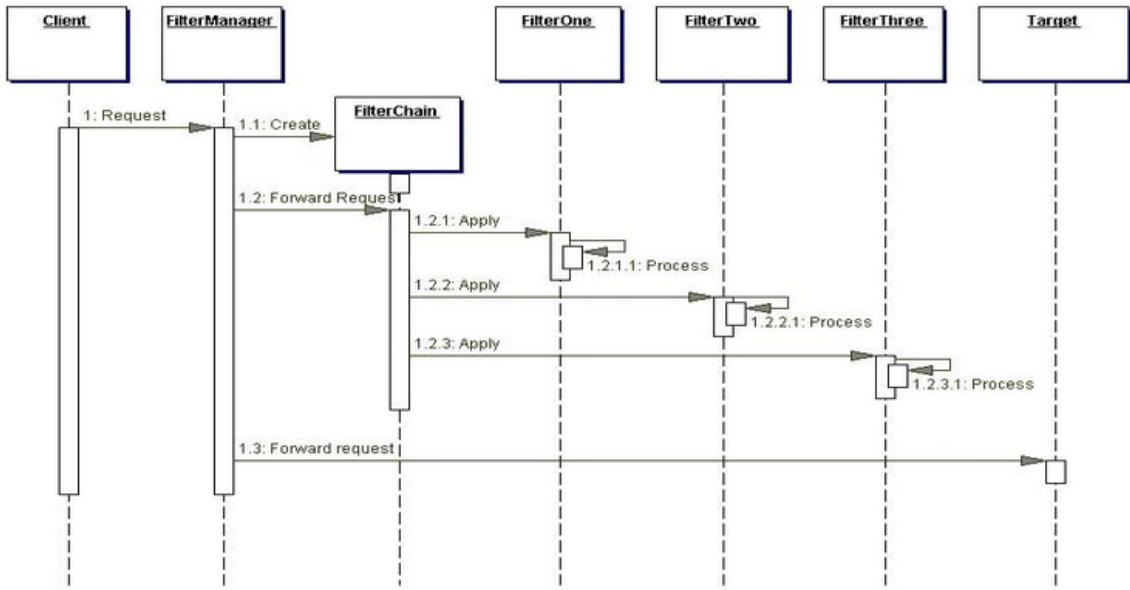


Figura 1.43: Ejemplo de diagrama de secuencias del estilo arquitectónico *Filtros de intercepción*. [Fuente: <https://www.oracle.com/technetwork/java/interceptingfilter-142169.html>.]

1.4. Patrones o estilos arquitectónicos

Un *estilo arquitectónico* (o patrón arquitectónico) define tipos de componentes y conectores y un conjunto de restricciones sobre la forma en la que pueden combinarse estos elementos, en el nivel más alto de abstracción de un sistema software. Una definición más formal es:

Un *patrón arquitectónico* expresa un esquema de organización estructural fundamental en un sistema software. Proporciona un conjunto de subsistemas predefinidos, especificando sus responsabilidades, e incluye reglas y guías para organizar las relaciones entre ellos.⁴⁷

Algunos de ellos se han intentado clasificar, como puede apreciarse en la Figura 1.44. Esta clasificación puede ayudar a entender los estilos y relacionar unos con otros.

⁴⁷Buschmann et al., *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*.

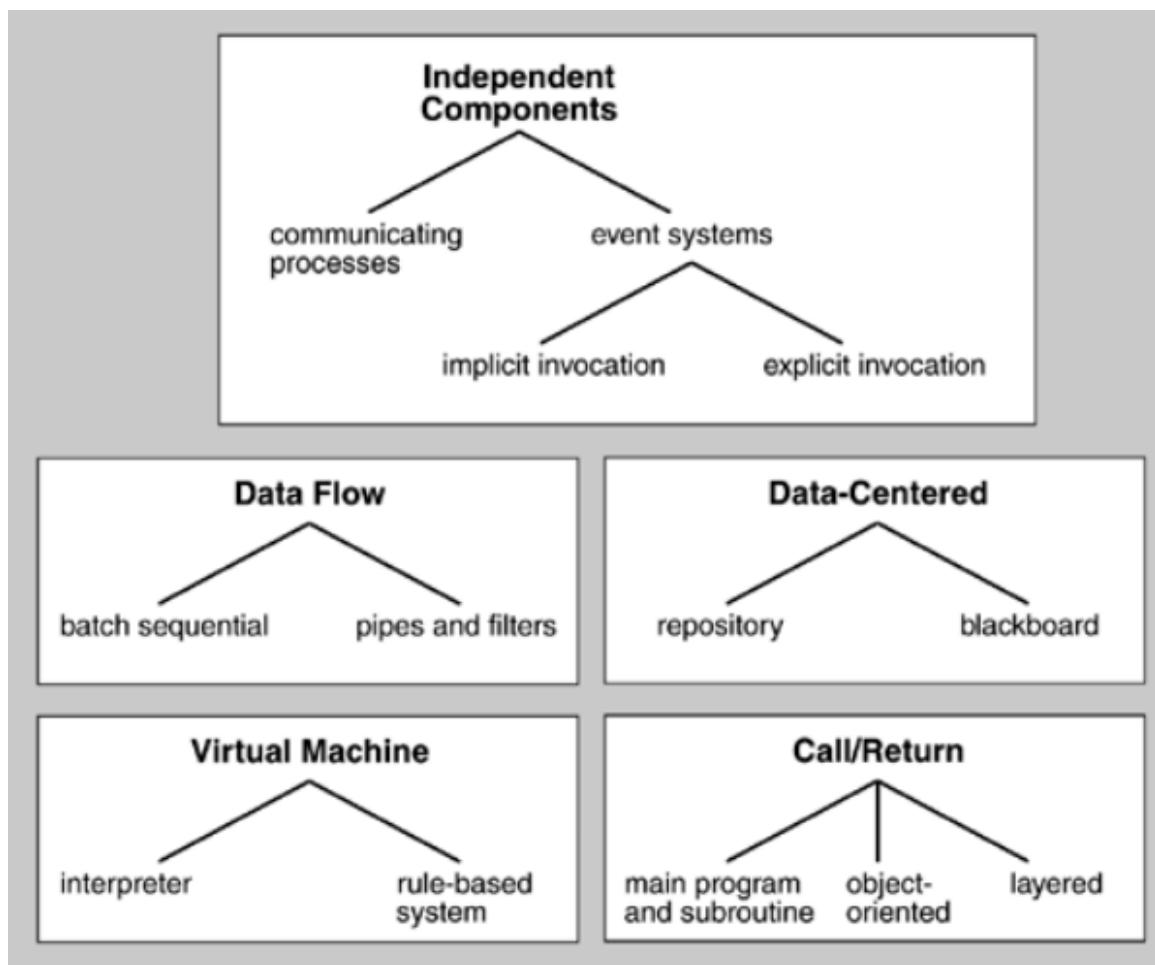


Figura 1.44: Clasificaciones de distintos estilos arquitectónicos *Control de procesos* para el sistema de control de la velocidad de crucero. [Fuente: (Len Bass, Paul Clements y Rick Kazman, *Software Architecture in Practice (2nd Edition)* [Addison-Wesley Professional, 2003])]

1.4.1. Estilos de Flujo de Datos: el estilo *Tubería y filtro*

Los estilos de Flujo de Datos «enfatizan la reutilización y la modificabilidad. Es apropiada para sistemas que implementan transformaciones de datos en pasos sucesivos.».⁴⁸

En el estilo *Tubería y filtro* los componentes son los *filtros* (filter) y los conectores son las *tuberías* (pipes). Las tuberías transportan la corriente de datos (data stream). El filtro recibe una corriente de datos de entrada y los va procesando de forma incremental, realizando con ellos una transformación y empezando a poner los datos ya transformados en la salida aún cuando todavía no haya consumido todo los datos de entrada que le llegan. Un ejemplo aparecen en la Figura 1.45.

⁴⁸Carlos Reynoso y Nicolás Kicillof, *Estilos y patrones en la estrategia de arquitectura de Microsoft.*, informe técnico (Universidad de Buenos Aires, 2004), <http://biblioteca.udgvirtual.udg.mx/jspui/handle/123456789/940>.

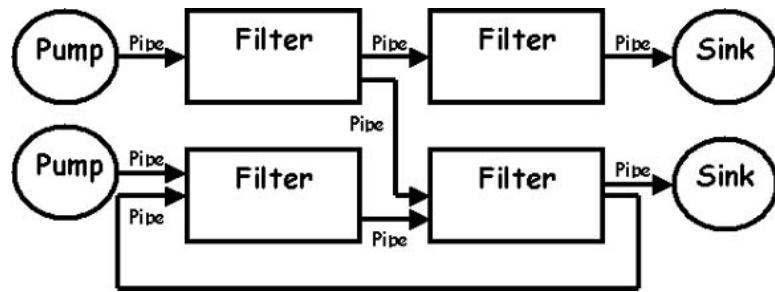


Figura 1.45: Estructura del estilo arquitectónico *Tubería y filtro*. [Fuente: (Garfixia, *Pipe-And-Filter*, visitado 4 de marzo de 2020, http://www.dossier-andreas.net/software_architecture/pipe_and_filter.html)]

Invariantes del estilo

- Los filtros son entidades independientes, no comparten su estado con otros filtros
- Los filtros no conocen a sus filtros vecinos, tan sólo pueden imponer restricciones a los datos que le entran y garantizar que cumple con restricciones a los datos que saca como salida. Ni siquiera debería afectar a la salida final del sistema la forma en la que cada filtro realizan el procesamiento incremental de los datos

Algunos subestilos o tipos más específicos del estilo *Tubería y filtro* son:

- Estilo *Tubería lineal* (pipelines).- Todos los filtros están en una única secuencia (ver Figura 1.46)
- Estilo *Tubería limitada*.- Restringe la cantidad máxima de datos que pueden estar en un momento dado en una tubería
- Estilo *Secuencial por lotes* (sequential batch).- Es un caso extremo en el que cada filtro procesa todos los datos de entrada en conjunto, como una única entidad. En este caso las tuberías realmente no tienen sentido porque no se requiere una corriente de datos. Realmente éste se considera un estilo independiente

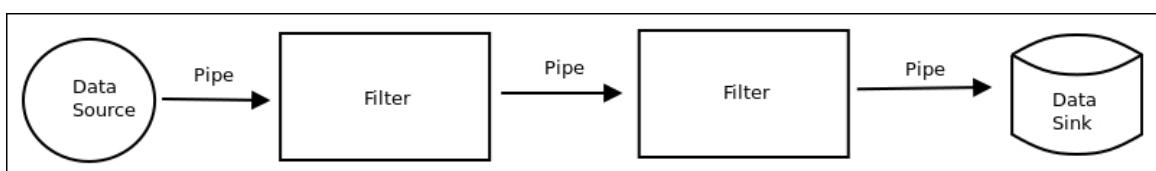


Figura 1.46: Estructura del estilo arquitectónico *Tubería y filtro*. [Fuente: (Anand Balachandran Pillai, *Software Architecture with Python* [Packt Publishing, 2017], <https://learning.oreilly.com/library/view/software-architecture-with/9781786468529/ch08s04.html>)]

El mejor ejemplo de este estilo lo proporciona Unix en su intérprete de comandos (shell). Para ello proporciona una notación que representa las tuberías, es decir, que conecta los

componentes (el carácter “|”) junto con mecanismos de ejecución para implementar las tuberías.

Por ejemplo, para obtener el total de ficheros en el directorio actual que contienen la palabra “Maze” en su nombre:

```
ls | grep Maze | wc -l
```

Otros ejemplos se dan en el procesamiento de señales o en la programación paralela.

Ventajas

- Diseño fácil: Los sistemas con este estilo son muy fáciles de ser entendidos
- Reusabilidad: Se pueden construir filtros por la unión de dos filtros, siempre que haya acuerdo en la información que pasa de uno a otro
- Mantenibilidad, incluyendo mejoras: Se pueden añadir nuevos filtros o reemplazar los que ya existen
- Permiten análisis especializado, como rendimiento y análisis de interbloqueos
- Concurrencia: cada filtro puede realizar una tarea y actuar en paralelo con otros

Inconvenientes

- No permiten proceso interactivo, sólo programación por lotes
- Bajo rendimiento: Si no se programan bien se pueden producir cuellos de botella en los filtros más lentos
- Complejidad en la programación: para que el rendimiento sea alto, requieren de una programación compleja

1.4.2. El estilo *Abstracción de Datos y Organización OO*

Los estilos de Llamada y Retorno «enfatizan la modificabilidad y la escalabilidad. Son los estilos más generalizados en sistemas en gran escala.»⁴⁹ Algunos ejemplos son las arquitecturas basadas en componentes, las orientadas a objetos (que se ven a continuación), el estilo basado en eventos que se verá en la siguiente Sección 1.4.4, la arquitectura Modelo-Vista-Controlador (Sección 1.4.3) y el estilo en capas (Sección 1.4.5).

El estilo *Abstracción de datos y organización OO* transparenta las características modulares cuando se usa un lenguaje de programación modular u *OO*. Así, en este estilo, la representación de los datos y las operaciones que éstos pueden hacer se encapsulan: Los componentes son instancias de tipos abstractos de datos (TADs) en lenguajes modulares u objetos en lenguajes *OO*. Los conectores son las vías de comunicación entre los componentes (asociaciones, en terminología *OO*). Una conexión concreta se hace operativa cuando

⁴⁹Reynoso y Kicillof, *Estilos y patrones en la estrategia de arquitectura de Microsoft*.

se realiza entre ellos alguna llamada o invocación (o envío de mensajes, en terminología **OO**) a procedimientos o funciones (o métodos, en terminología **OO**). A los componentes también se les llama gestores por ser los encargados de preservar la integridad de los datos que contienen.

En la Figura 1.47 se muestra un ejemplo. “op” es la invocación (procedure call) que se hace para conectar un objeto con otro.

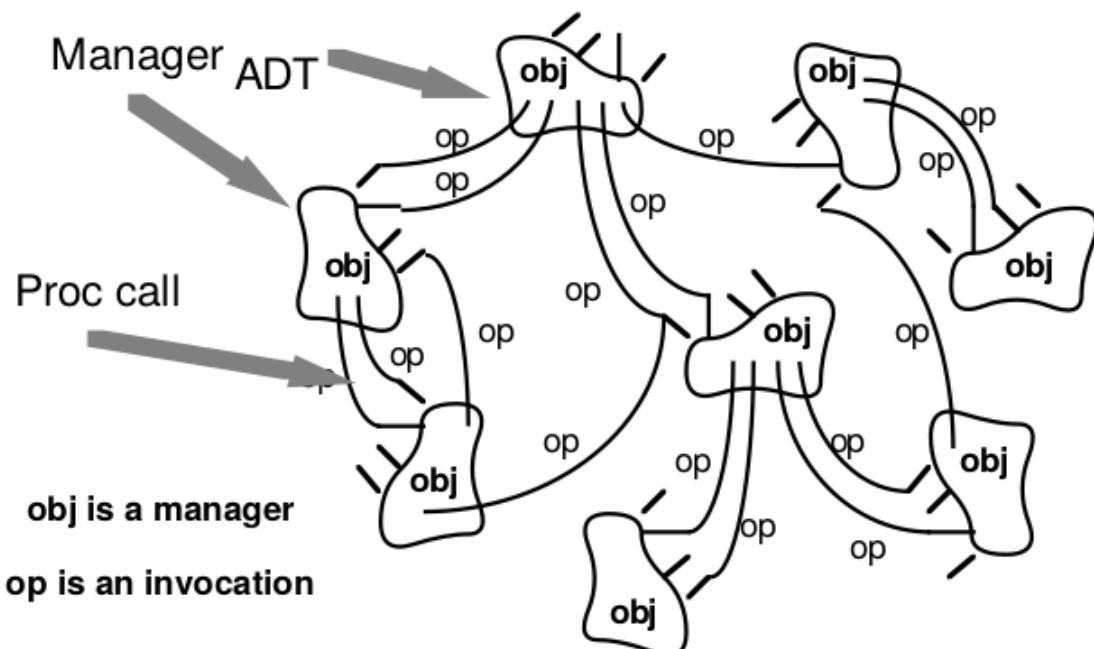


Figura 1.47: Ejemplo del estilo arquitectónico *Abstracción de Datos y Organización **OO***. [Fuente: (Mary Shaw y David Garlan, *Software Architecture* [New Jersey: Prentice Hall, 1996], pg. 23)]

Invariantes del estilo

- Encapsulamiento: Datos y operaciones sobre ellos están encapsulados
- El objeto es responsable de preservar la integridad de su estado
- Ocultación: La implementación de las operaciones están ocultas a otros objetos. Los datos también pueden ocultarse

Ventajas

- Gracias al ocultamiento de las operaciones es posible cambiar la implementación de éstas sin que afecte a sus clientes
- Gracias al encapsulamiento, los diseñadores pueden descomponer la resolución de un problema como un conjunto de agentes con responsabilidades específicas que cooperan para la resolución del mismo

Inconvenientes

- Los componentes deben conocerse entre ellos para poder interaccionar (piénsese como alternativa en el estilo *Tubería y filtro*)
- Puede haber efectos colaterales entre objetos que están conectados entre sí directa o indirectamente

1.4.3. El estilo *Modelo-Vista-Controlador* (MVC)

Fue descrito por primera vez para su aplicación en Smalltalk.⁵⁰ La idea fundamental es separar la funcionalidad del sistema (*lógica del negocio* o *dominio de la aplicación*) de la interfaz de usuario. En la actualidad está ampliamente extendido. Ejemplos de uso son Smalltalk y Ruby on Rails.

Se compone de tres elementos:

- Modelo: Conjunto de clases que representan la lógica de negocio de la aplicación (clases deducidas del análisis del problema). Encapsula la funcionalidad y el estado de la aplicación.
- Vista: Representación de los datos contenidos en el modelo. Para un mismo modelo pueden existir distintas vistas.
- Controlador: Es el encargado de interpretar las órdenes del usuario. Mapea la actividad del usuario con actualizaciones en el modelo. Puesto que el usuario ve la vista y los datos originales están en el modelo, el controlador actúa como intermediario y mantiene ambos coherentes entre sí.

⁵⁰Trygve Reenskaug, *A note on DynaBook requirements*, informe técnico (Xerox PARC, 1979), <http://folk.uio.no/trygver/1979/sysreq/SysReq.pdf>.

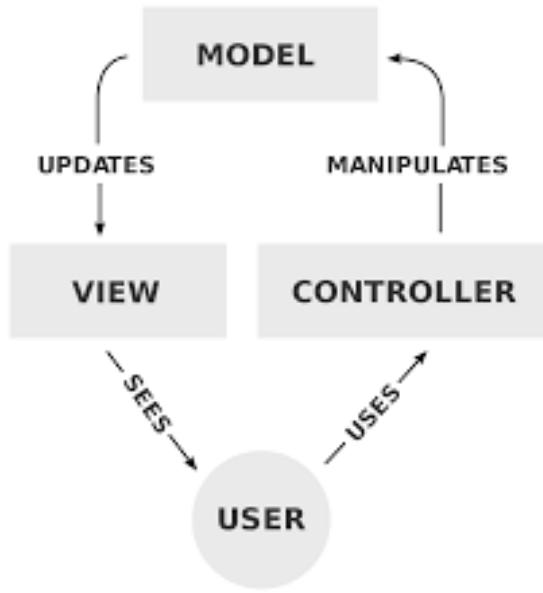


Figura 1.48: Estructura del estilo arquitectónico **MVC** de Controlador ligero (ver más abajo). [Fuente: <https://en.wikipedia.org/wiki/Model%20view%20controller>]

Ventajas

- El desarrollo es más sencillo y limpio.
- Facilita la evolución por separado de vista y modelo.
 - Cualquier modificación que afecte al dominio de la aplicación, implica una modificación sólo en el modelo. Las modificaciones a las vistas no afectan al modelo, simplemente se modifica la representación de la información, no su tratamiento.
- Incrementa la reutilización y la flexibilidad.

Inconvenientes

- Curva de aprendizaje muy pendiente, pues muchas veces este estilo combina tecnologías múltiples que los desarrolladores
- Puede añadir complejidad superflua si la aplicación no lo requiere

Variantes Hay principalmente dos variantes en este estilo:

- Controlador ligero o Modelo activo: La actualización de las vistas es hecha directamente desde el modelo (Figura 1.48). Para mantener la independencia de ambas, en esta opción es necesario el uso a su vez del estilo *Basado en Eventos*, estilo que veremos a continuación, y que es equivalente al patrón de diseño *Observador* que ya vimos.

- Controlador pesado o Modelo pasivo: el controlador conoce las vistas e interactúa con ellas para notificarles que se han hecho cambios en el modelo de forma que las vistas pedirán los datos al modelo (lo cuál puede hacerse directa o indirectamente).

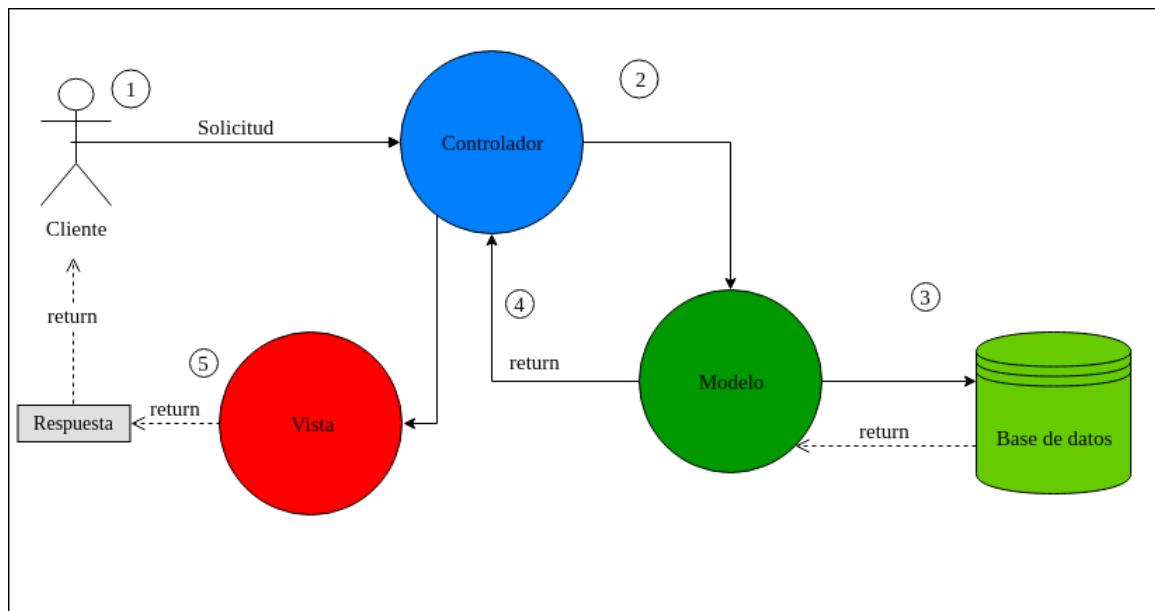


Figura 1.49: Estructura del estilo arquitectónico **MVC** de Controlador pesado. [Fuente: <https://articulosvirtuales.com/articles/educacion/que-es-el-modelo-vista-controlador-mvc-y-como-funciona>]

Un ejemplo de implementación del estilo **MVC** del controlador ligero con el estilo de eventos de invocación implícita es el que hace la librería Java SWING. En ella, se ha pasado del clásico **MVC** a una *arquitectura de modelo separable* (separable model architecture) en la que el controlador y la vista están fuertemente acoplados en una única clase llamada **UIObject** (o **UIDelegate**) (Ver Figura 1.50)⁵¹.



Figura 1.50: Implementación del estilo **MVC** hecha por Java SWING.

⁵¹El elemento **UI Manager** es usado para almacenar información general a la apariencia de todos los componentes (color de fondo por defecto, tipo de letra por defecto, tipos de bordes, etc.), estando todos manejados por el mismo **UI Manager**.

Un ejemplo de utilización del estilo de controlador pesado es el que realiza el entorno de trabajo Angular JS para desarrollo Web, escrito en JavaScript (ver Figura 1.51).

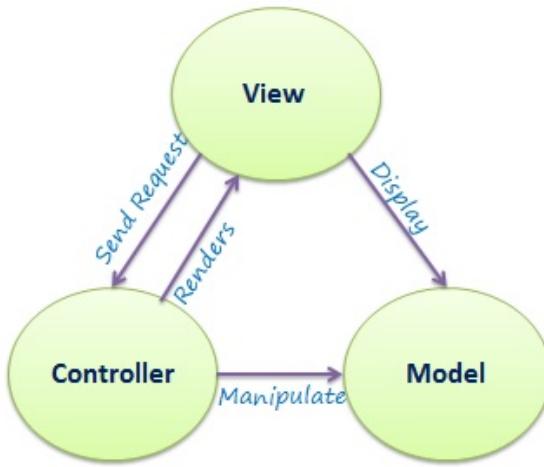


Figura 1.51: Estructura del estilo arquitectónico **MVC** pesado implementado por Angular JS. [Fuente: <https://w3tutoriels.com/angularjs/angularjs-mvc/>]

Existe una alternativa intermedia. Los controladores son suficientemente pesados para interactuar con las vistas, es decir, no lo hace el modelo directamente. Sin embargo, son los más ligeros posible, en el sentido de que todo lo que pueda hacer el modelo, se lo dejan al modelo, y de que tienen responsabilidades simples, pareciéndose al modelo en ese sentido. Este es el caso de Ruby on Rails (RoR), que tiene la máxima de *Fat models, skinny controllers*. Así, para cada clase del modelo, habrá un controlador, que hará lo mínimo: manejar el tráfico de datos, básicamente manejar la petición del usuario según la ruta, pasarlal al modelo y devolver al usuario el resultado (vista).



RoR 1.4.1. ¡CUIDADO!: controladores pesados

Cuando tenemos controladores pesados en RoR, es porque están haciendo tareas que pertenecen al modelo y no seguimos las convenciones de Rails (la primera es: *Convention Over Configuration*).

A continuación podemos ver un ejemplo de un controlador (fichero `projects_controller.rb`) para la clase **Project**:

```

class ProjectsController < ApplicationController

  before_action :set_project, only: [:show, :edit, :update, :destroy]

  # GET /projects or /projects.json
  def index
    @projects = Project.all
  end

```

```
# GET /projects/1 or /projects/1.json
def show
end

# GET /projects/new
def new
  @project = Project.new
end

# GET /projects/1/edit
def edit
end

# POST /projects or /projects.json
def create
  @project = Project.new(project_params)

  respond_to do |format|
    if @project.save
      format.html { redirect_to project_url(@project), notice: "Project was successfully created." }
      format.json { render :show, status: :created, location: @project }
    } else
      format.html { render :new, status: :unprocessable_entity }
      format.json { render json: @project.errors, status: :unprocessable_entity }
    end
  end
end
```

```
# PATCH/PUT /projects/1 or /projects/1.json
def update
  respond_to do |format|
    if @project.update(project_params)
      format.html { redirect_to project_url(@project), notice: "Project was successfully updated." }
      format.json { render :show, status: :ok, location: @project }
    else
      format.html { render :edit, status: :unprocessable_entity }
      format.json { render json: @project.errors, status: :unprocessable_entity }
    end
  end
end

# DELETE /projects/1 or /projects/1.json
def destroy
  @project.destroy

  respond_to do |format|
    format.html { redirect_to projects_url, notice: "Project was"
```

```
    successfully destroyed." }
    format.json { head :no_content }
  end
end

private
  # Use callbacks to share common setup or constraints between actions.
  def set_project
    @project = Project.find(params[:id])
  end

  # Only allow a list of trusted parameters through.
  def project_params
    params.require(:project).permit(:name, :team, :info)
  end
end
```

1.4.4. El estilo *Basado en Eventos*

Una alternativa a la llamada a procedimientos propia del estilo *Abstracción de datos y organización OO* (a partir de ahora, estilo *OO*, para simplificar) es la posibilidad de no hacer una invocación directa a un procedimiento o función (un método en *OO*) sino de forma indirecta. Para ello, los componentes con métodos cuya invocación dependa del estado de otros componentes, deben suscribirse a una lista de partes interesadas en este otro componente. Cuando en este otro componente se produce un cambio de estado (evento), lo retransmite o anuncia a los subscriptores de la lista, o lo publica y otro componente recoge la publicación para transmitirla a los subscriptores. Los subscriptores reciben el anuncio y realizan las operaciones necesarias a partir de la información que les ha sido comunicada. Este estilo también está considerado dentro del grupo de estilos *Peer-to-Peer*.⁵²

Variantes Hay dos variantes dentro de este estilo:

- Invocación implícita (estilo *Manejador de Eventos* o *Publicar/subscribir*).- El componente que anuncia un evento no conoce quiénes son sus clientes a la escucha. La forma de comunicación es mediante un tercer componente intermediario (manejador de evento) que es el que recoge los cambios publicados y sí conoce a los componentes que están interesados para retransmitírselos. Ejemplos de este estilo son usados por la librería SWING para la GUI en Java o por Flutter. En el caso de SWING, los que manejan el evento son objetos de la clase **ActionListener**. En Flutter, para los eventos con varios subscriptores (listeners) la suscripción se puede hacer a través de la clase **Listenable**.

⁵²Reynoso y Kicillof, *Estilos y patrones en la estrategia de arquitectura de Microsoft*.

- Invocación explícita (patrón *Observador*⁵³).- El componente en el que se produce el evento tiene él mismo la lista de subscriptores y les anuncia a todos el cambio que se ha producido.

En la Figura 1.52 puede verse la diferencia entre el subestilo de *invocación explícita* u *Observador* (también un patrón de diseño) y el subestilo *Manejador de Eventos*.

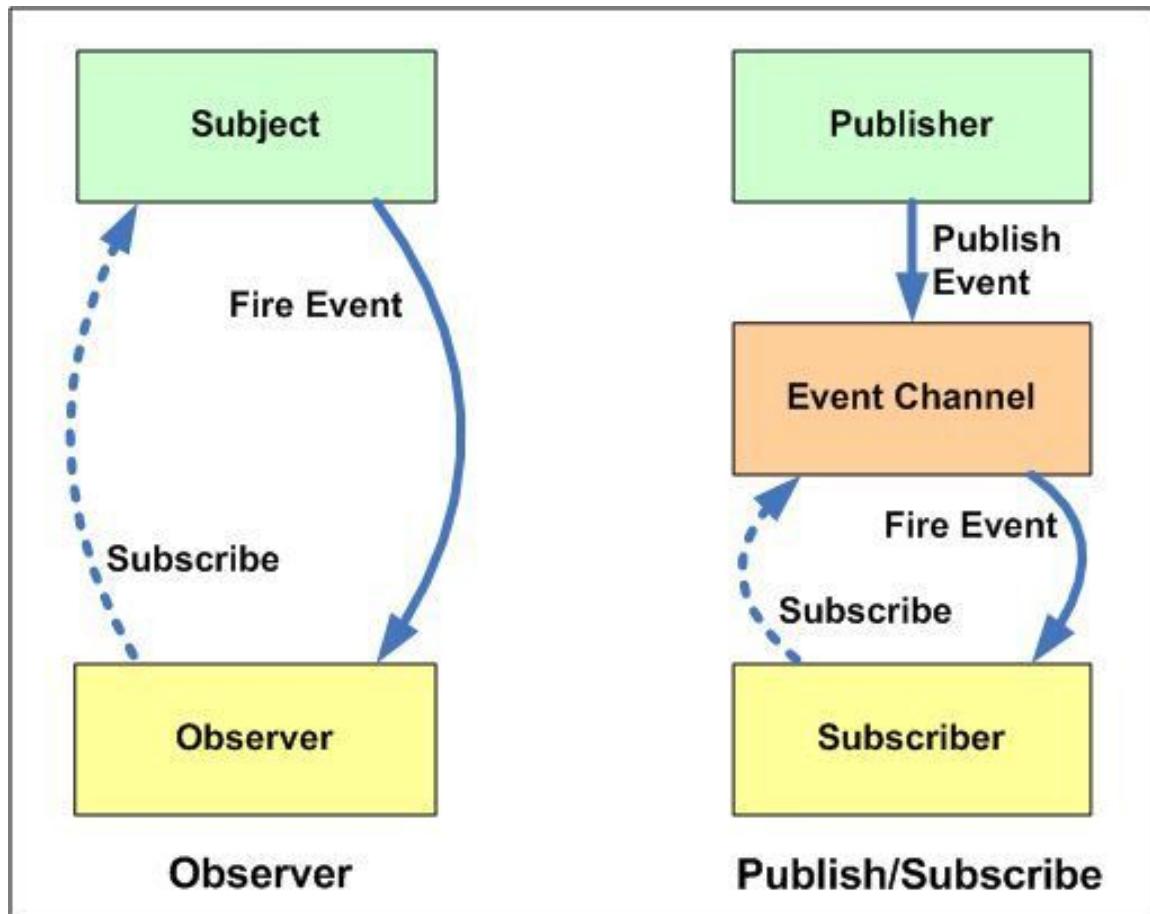


Figura 1.52: Diferencia entre los estilos arquitectónicos por invocación implícita (*Manejador de Eventos* o *Publicar/subscribir*) y el de invocación explícita (*Observador*). [Fuente: <https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c>]

```
public class Observable
{
    public ActionEvent<Object, Object> onStateChange;

    public void eventRelease()
    {
```

⁵³Obsérvese que éste es también propuesto como patrón de diseño. Éste es un ejemplo de la fuerte relación que hay entre ellos, hasta tal punto de que muchos autores no hacen tal distinción, considerando además que diferentes patrones pueden aplicarse sobre un mismo sistema de forma jerárquica.

```

        // do something
        // choose appropriate args
        onStateChange.invoke(null, null);
    }

    /* versus *****/
    Observator[] observers;
    public void informObservators()
    {
        foreach (Observer observer in observers)
            // choose appropriate args
            observer.updateState(null, null);
    }
}

public class Observer
{
    public Observer(Observable observable)
    {
        observable.onStateChange += updateState;
    }

    /* versus *****/
    public void updateState(Object arg1, Object arg2)
    {

    }
}

```

Un ejemplo de uso del estilo de invocación implícita son los depuradores software. Cuando el depurador se para en un punto de interrupción (breakpoint), se anuncia esta parada. Los que reciben el anuncio operan en consecuencia, por ejemplo, el editor de textos para ponerse en la línea de parada y la ventana con el estado de las variables se actualiza.

Ventajas

- Reusabilidad: Pueden añadirse componentes al sistema sólo añadiéndolos a la lista de subscriptores
- Evolución más sencilla del sistema: Los componentes pueden reemplazarse (cambiando incluso su interfaz) sin que ellos afecte a las interfaces del resto de componentes con los que se relaciona (pues lo hacen de forma indirecta)

Inconvenientes

- Pérdida de control: Los componentes no saben realmente cómo afecta lo que hacen en otros componentes, ni siquiera el orden en el que los cambios se producen en los otros componentes (sobre todo en el estilo *Manejador de eventos*)
- Difícil comprobación de integridad: Derivada de la pérdida de control también se hace difícil establecer las condiciones de ejecución de una operación concreta, frente

a la tradicional llamada a procedimientos (estilo *OO* o estilo *Organización programa principal/subrutinas* (ver más abajo) para los que bastan las precondiciones y postcondiciones de una función o procedimiento para conocer su comportamiento en un momento determinado.

1.4.5. El estilo *Sistema por Capas*

En este estilo el sistema se organiza por capas, de forma que cada capa sirve a la capa superior y es cliente de la capa inferior. Cuando una capa sólo puede ver a las adyacentes podemos hablar de que se trata de una máquina virtual de cara a su capa cliente. La Figura 1.53 proporciona un ejemplo con sólo tres capas, mientras que la Figura 1.54 proporciona un diagrama genérico.

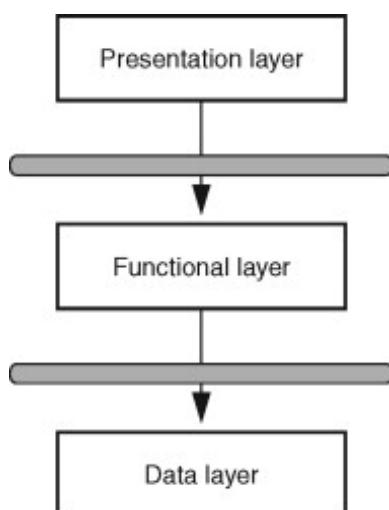


Figura 1.53: Estructura del estilo arquitectónico *Sistema por Capas* con sólo tres capas. [Fuente: (Heinz Züllighoven, *Object-Oriented Construction Handbook* [EE.UU.: Science Direct, 2005])]

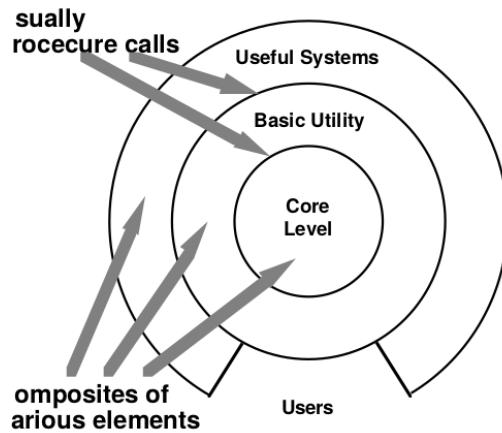


Figura 1.54: Estructura del estilo arquitectónico *Sistema por Capas* genérico. [Fuente: (Mary Shaw y David Garlan, *Software Architecture* [New Jersey: Prentice Hall, 1996], pg. 25)]

Un ejemplo de este estilo es la forma en la que interactúa todo el software de un ordenador (capas: firmware y sistema operativo, utilidades, aplicaciones de usuario). Otro son las capas de un Sistema de Gestión de Bases de Datos (SGBD) (capas: física –gestor de datos y metadatos en disco–, lógica –gestor de operaciones de consulta y actualización de datos como entidades provistas de significado para el usuario–, de aplicación: gestión de vistas y solicitudes de usuario).

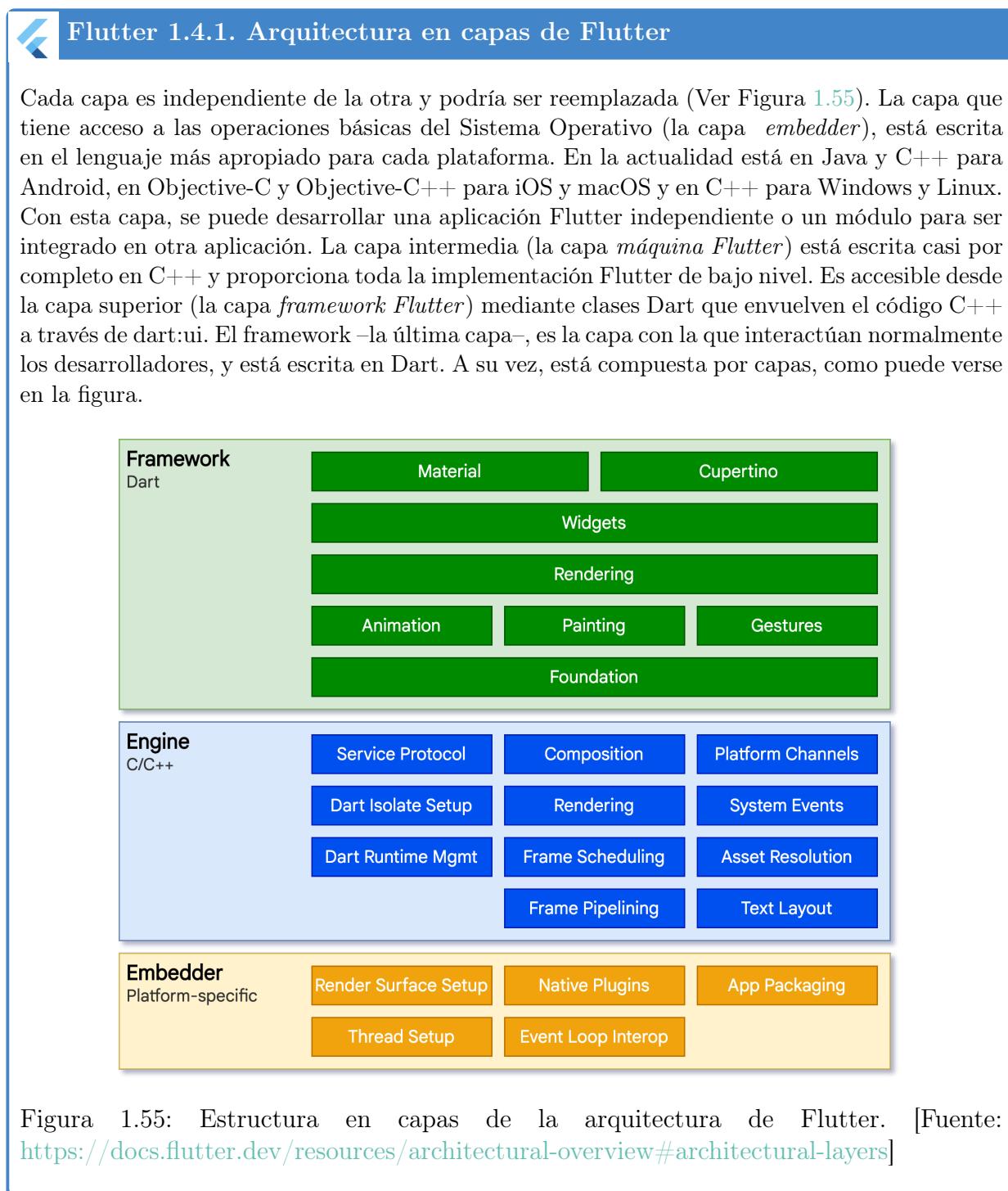
Ventajas

- Soportan diseños basados en niveles de abstracción incremental.
- Reusabilidad: permiten la sustitución de la implementación de una capa por otro código, siempre que se mantenga su interfaz (igual que en el estilo [OO](#)).

Inconvenientes

- Difícil adaptación: No todos los sistemas pueden ser concebidos de forma jerárquica. Incluso si lo hacen pueden perder eficiencia frente a un estilo que permita mayor cohesión entre componentes.

Otro ejemplo es la arquitectura de Flutter, como puede verse en la Figura [1.55](#).



1.4.6. Estilos Centrados en Datos. El estilo *Repositorio*

Los estilos Centrados en Datos «enfatizan la integrabilidad de los datos. Se estiman apropiados para sistemas que se fundan en acceso y actualización de datos en estructuras de

almacenamiento.».⁵⁴

El estilo *Repositorio* está compuesto por un componente como estructura central de datos (almacén de datos o repositorio) y el resto de componentes, externos, que operan sobre él.

Variantes Se considera que existen dos variantes:

- Estilo *Repositorio Básico*: Cuando es una petición externa hacia un componente externo la que dispara una petición para que se ejecute un proceso concreto de ese componente que a su vez solicitará información al componente central. Un ejemplo es el uso de una base de datos distribuida.
- Estilo *Pizarra* (blackboard): Es el propio estado en el que se encuentra el componente central el que dispara el proceso a realizarse en un componente externo o *fuente de conocimiento* (knowledge source). Se utiliza en problemas para los que no existen estrategias para encontrar soluciones deterministas.

En una Pizarra, existen varios subsistemas especializados (*fuentes de conocimiento*, del inglés *knowledge sources*, *ks*) cuyo conocimiento es unido para encontrar una posible solución parcial o aproximada. Un ejemplo de la relación entre la pizarra y las *ks* en el estilo *Pizarra* puede verse en la Figure 1.56.

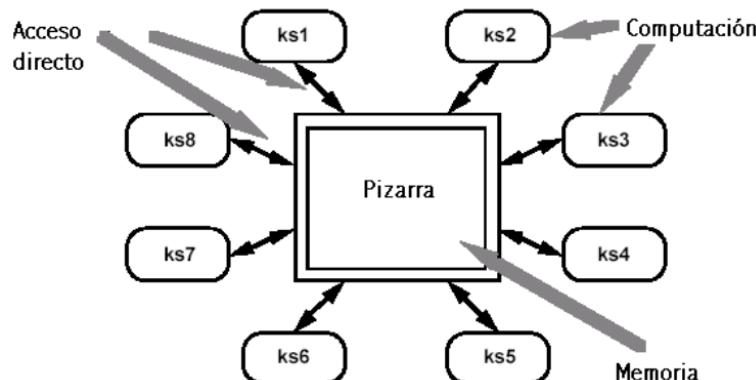


Figura 1.56: Relación entre la pizarra y las fuentes de conocimiento en el estilo arquitectónico *Pizarra*. [Fuente: (Mary Shaw y David Garlan, *Software Architecture* [New Jersey: Prentice Hall, 1996], pg. 26)]

La Figura 1.57 usa un diagrama de clases (diagrama a nivel de diseño, no de arquitectura), para implementar el estilo Pizarra combinándolo con el estilo *OO*. En esta implementación, todos los componentes del estilo se programan como clases que se relacionan entre sí, y se utiliza un tercer componente. Se trata del componente de control, que permiten seguir y centralizar una estrategia para llegar a la solución.⁵⁵

⁵⁴Reynoso y Kicillof, *Estilos y patrones en la estrategia de arquitectura de Microsoft*.

⁵⁵Buschmann et al., *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*.

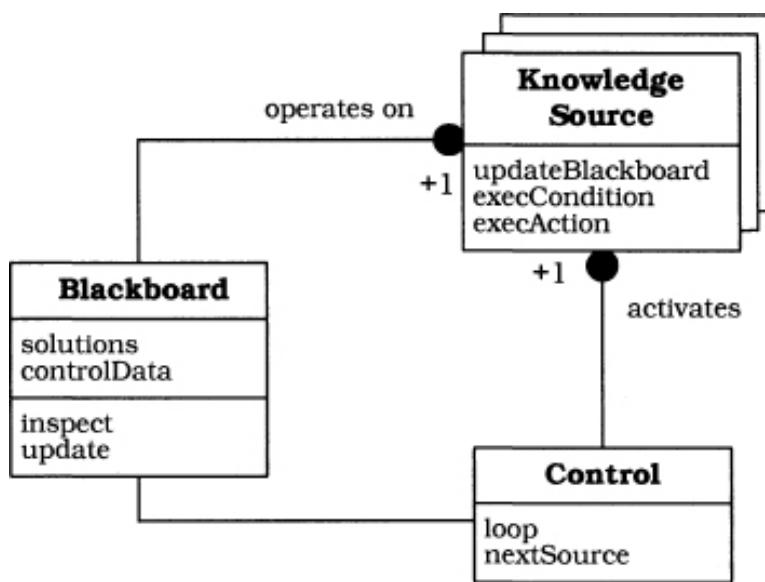


Figura 1.57: Diagrama de clases para representar el estilo arquitectónico *Pizarra*. [Fuente: (Frank Buschmann et al., *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns* [Wiley Publishing, 1996], pg. 79, <https://learning.oreilly.com/library/view/pattern-oriented-software-architecture/9781118725269/>)]

Un ejemplo de aplicación del estilo *Pizarra* es en el reconocimiento de voz, donde la solución no es determinista y donde cada fuente de conocimiento requiere un tipo distinto de información (ondas acústicas, fonemas, palabras, sintagmas, etc.).

1.4.7. Estilos de Código Móvil. El estilo *Intérprete*

Los estilos de Código Móvil enfatizan la portabilidad. Algunos ejemplos son los intérpretes, los sistemas basados en reglas y los procesadores de lenguaje de comandos.⁵⁶

El estilo *Intérprete* permite crear una máquina virtual que sirva de puente entre el nivel de computación básico que está disponible (programa objeto) para interactuar con el hardware y el nivel lógico que es entendido por la semántica de una aplicación.

Usualmente está formado por cuatro componentes:

- El motor que interpreta (o intérprete en sí)
- El contenedor con el pseudocódigo a ser interpretado
- Una representación del estado en el que se encuentra el motor de interpretación
- Una representación del estado en el que se encuentra el programa que está siendo simulado

⁵⁶Reynoso y Kicillof, *Estilos y patrones en la estrategia de arquitectura de Microsoft*.

La Figure 1.58 muestra la estructura del estilo *Intérprete*.

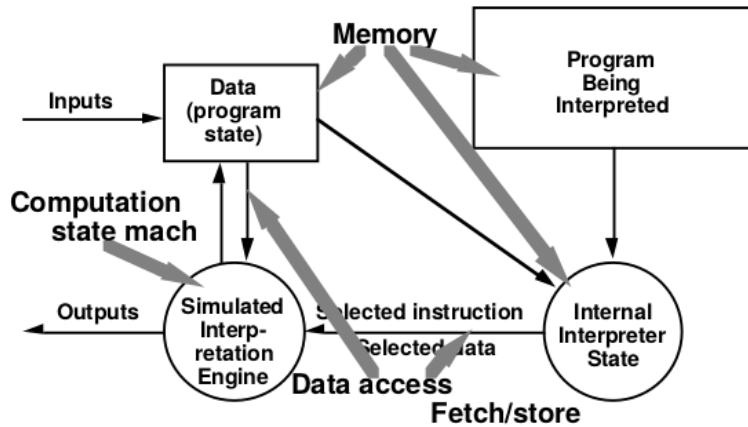


Figura 1.58: Estructura del estilo arquitectónico *Intérprete*. [Fuente: (Mary Shaw y David Garlan, *Software Architecture* [New Jersey: Prentice Hall, 1996], pg. 27)]

Algunos ejemplos de lenguajes que usan intérpretes (lenguajes interpretados) actuales son Python, PHP y R.

Un ejemplo de intérprete más sofisticado son las máquinas virtuales, usadas por ejemplo para Java (JVM) o para Ruby (RubyVM) y disponibles para cualquier plataforma. En este caso, el código de entrada son los bytecodes (ficheros .class resultantes de la compilación de los .java o los .rb respectivamente).

1.4.8. Estilos heterogéneos. Estilo *Control de procesos*

Su propósito es el de mantener las propiedades de salida (variables controladas) de un proceso en un nivel o cercano a un nivel de referencia (set point). Se usa sobre todo en el entorno industrial.

Variantes Existen algunas variantes:

- *Ciclo abierto* (Figure 1.59).- En muy pocos casos, cuando todo el proceso es completamente predecible, no es necesario vigilar el proceso (controlar el estado de las variables y reaccionar en consecuencia).
- *Ciclo cerrado* (Figure 1.60).- Es necesario supervisar el sistema para corregir la salida según el cambio en los valores de las variables de entrada.
 - *Control de procesos retroalimentado* (Figure 1.61)
 - *Control de procesos preventivo* (Figure 1.62)

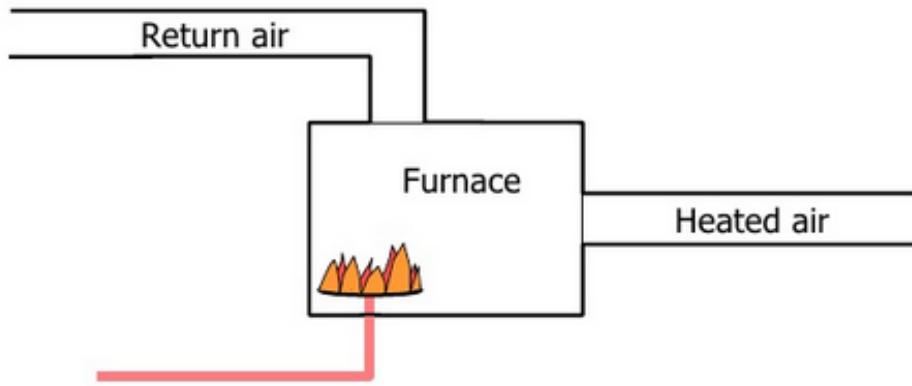


Figura 1.59: Ejemplo de sistema con estilo arquitectónico *Control de procesos de ciclo abierto*. [Fuente: (Mary Shaw y David Garlan, *Software Architecture* [New Jersey: Prentice Hall, 1996], pg. 29)]

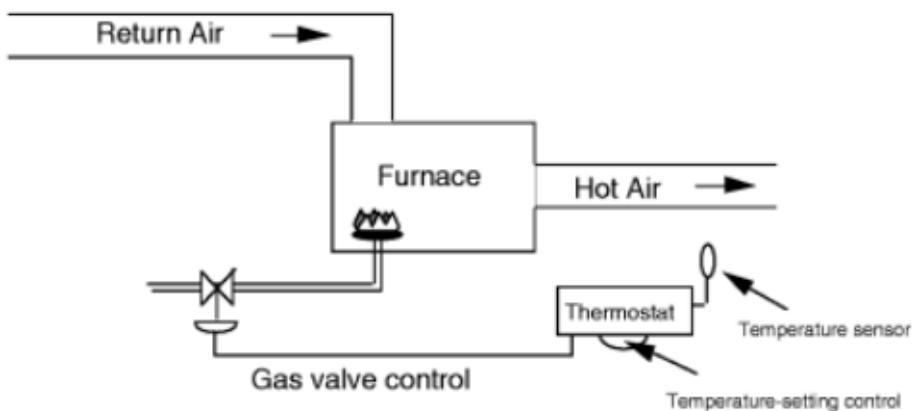


Figura 1.60: Ejemplo de sistema con estilo arquitectónico *Control de procesos de ciclo cerrado*. [Fuente: (Mary Shaw y David Garlan, *Software Architecture* [New Jersey: Prentice Hall, 1996], pg. 29)]

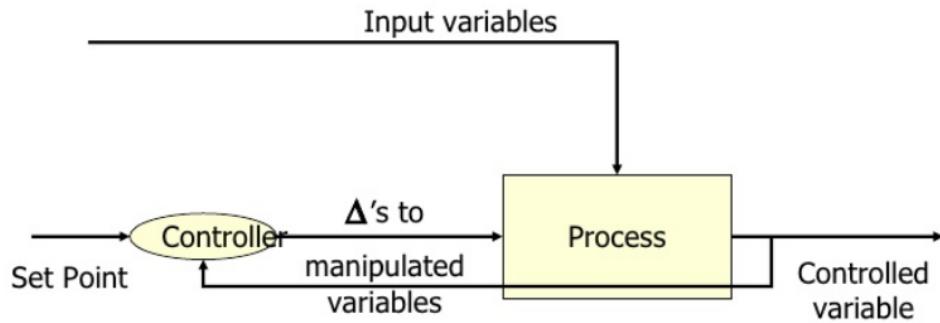


Figura 1.61: Estructura del estilo arquitectónico *Control de procesos retroalimentado*. [Fuente: (Mary Shaw y David Garlan, *Software Architecture* [New Jersey: Prentice Hall, 1996], pg. 29)]

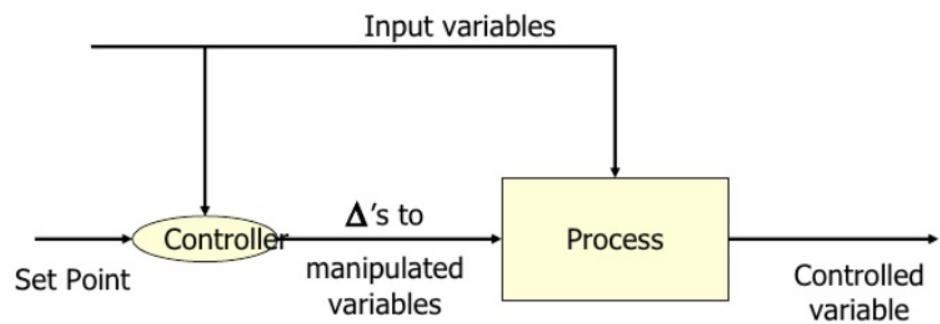


Figura 1.62: Estructura del estilo arquitectónico *Control de procesos preventivo*. [Fuente: (Mary Shaw y David Garlan, *Software Architecture* [New Jersey: Prentice Hall, 1996], pg. 30)]

La figura 1.64 muestra un diagrama para representar el control de procesos para un ejemplo del estilo arquitectónico *Control de procesos retroalimentado*: un sistema de control de la velocidad de crucero.

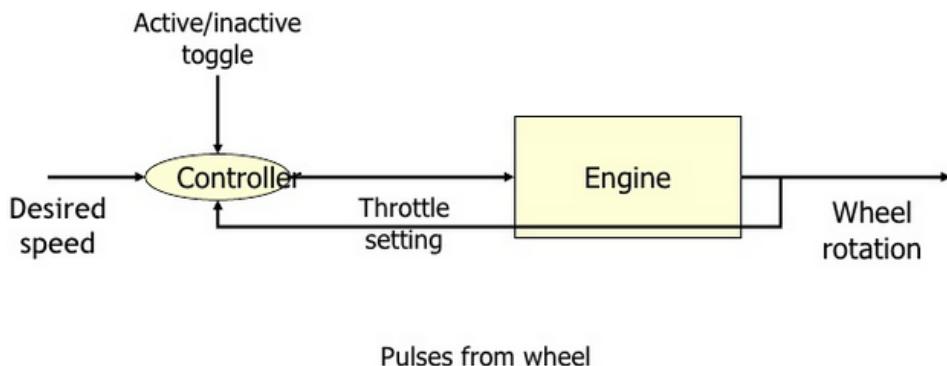


Figura 1.63: Diagrama que muestra el estilo arquitectónico *Control de procesos retroalimentado* para el sistema de control de la velocidad de crucero. [Fuente: (Greg Phillips et al., *Process Control Architectures*, Royal Military College of Canada, 1999, <https://www.slideshare.net/ahmad1957/process-control>)]

El sistema tiene dos elementos computacionales (puede ser software de uso dedicado, es decir, para ser usado en componentes electrónicos de uso específico y no en ordenadores de uso general, como la electrónica asociada al motor de un coche y la asociada a su sistema de control de velocidad de crucero): (1) el proceso a controlar, en este caso el motor (engine, en la figura) y (2) el algoritmo de control, que mantiene la velocidad de crucero (controller, en la figura). Por otro lado, se distinguen cuatro elementos de información:

1. Variable controlada: velocidad actual del coche
2. Variable manipulada (set point): posición de la palanca del sistema de control de la velocidad de crucero (throttle setting, en la figura)
3. Datos arrojados por el sensor de la variable controlada: revoluciones de la rueda (wheel pulses), que se convierte en velocidad por medio de un reloj que permite medir su frecuencia

La figura 1.64 muestra un diagrama arquitectónico (es decir, donde se muestra la estructura del sistema software al más alto nivel) para el ejemplo anterior del estilo arquitectónico *Control de procesos retroalimentado*, consistente en el sistema de control de la velocidad de crucero. En este caso se trata de un diagrama de componentes. La propuesta es de David Garlan.

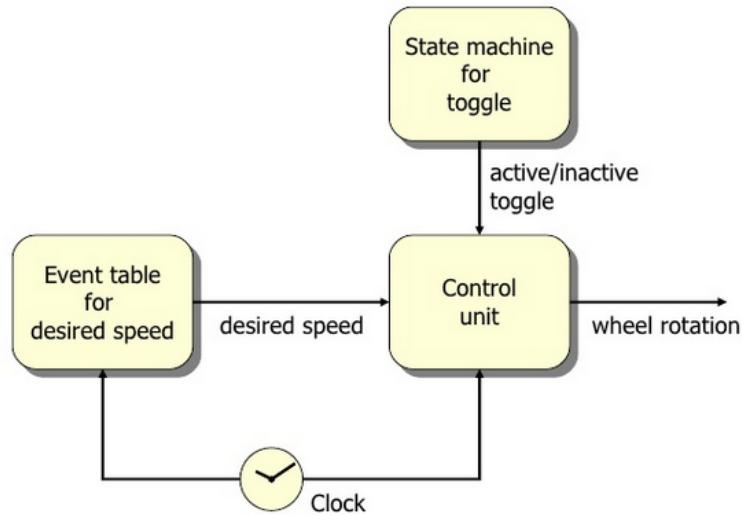


Figura 1.64: Diagrama de componentes. [Fuente: (Greg Phillips et al., *Process Control Architectures*, Royal Military College of Canada, 1999, <https://www.slideshare.net/ahmad1957/process-control>)]

La Figura 1.65 muestra un diagrama de contexto del sistema (es un diagrama donde todo el sistema es un bloque o caja negra).

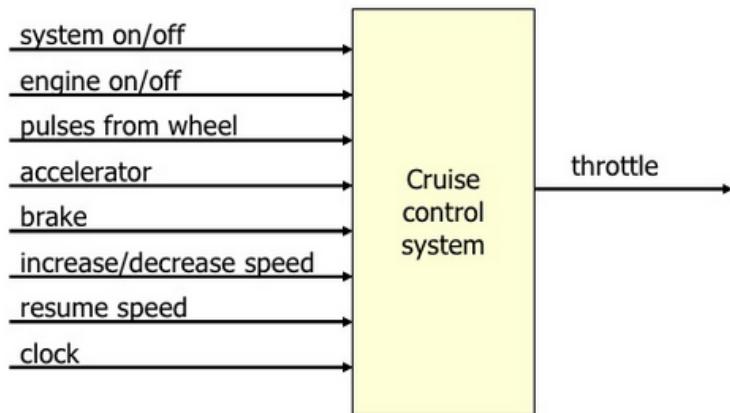


Figura 1.65: Diagrama de contexto del sistema de control de la velocidad de crucero. [Fuente: (Mary Shaw y David Garlan, *Software Architecture* [New Jersey: Prentice Hall, 1996], pg. 52)]

La figura 1.66 muestra un diagrama de diseño⁵⁷ del sistema de control de la velocidad de crucero, en donde se combina la arquitectura **OO** (estilo *Abstracción de datos y organización OO*) con la de *control de procesos retroalimentado*, de forma que tanto las partes del coche

⁵⁷Se trata de un diagrama funcional, en un nivel más bajo de abstracción, después del primer nivel, o nivel arquitectónico.

como las variables de entrada son codificadas como clases o atributos (ejemplo de solución propuesta según la arquitectura de Booch).

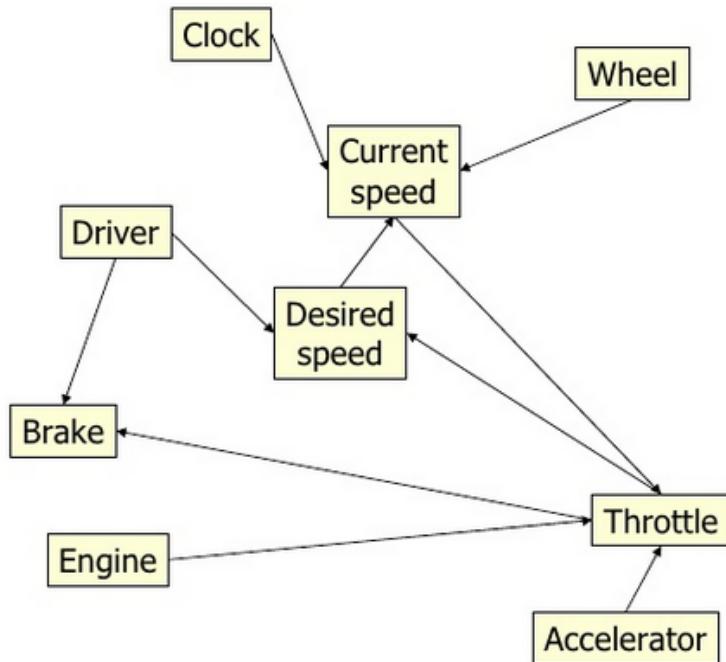


Figura 1.66: Diagrama de clases para el sistema de control de la velocidad de crucero. [Fuente: (Greg Phillips et al., *Process Control Architectures*, Royal Military College of Canada, 1999, <https://www.slideshare.net/ahmad1957/process-control>)]

1.4.9. Otros estilos arquitectónicos

Hay otros muchos paradigmas o estilos arquitectónicos, que pueden ser aplicados de forma simultánea. Algunos de ellas se describen a continuación:

- Estilo *Organización programa principal/subrutinas*.- Es el estilo que proyecta directamente la forma de funcionamiento de los lenguajes de programación que son usados por el sistema cuando éstos no permiten modularidad. Pertenece a la categoría de estilos de *Llamada y Retorno*.
- Estilo *Sistema de Transición de Estados*.- Utilizado, por ejemplo, en sistemas que definen estados y transiciones entre ellos para pasar de un estado a otro.
- Paradigma de *Programación reactiva*.- Es compatible con otros paradigmas de programación, como por ejemplo funcional e imperativa o procedural. Se refiere al uso de programación asíncrona para actualizar en tiempo real el estado de un sistema como consecuencia de otros cambios de estado ocurridos.

Un ejemplo es el uso de forma interna en los sistemas para la construcción de interfaces de usuario JavaScript React, de Facebook y el framework Flutter, de Google, que se inspiró en la primera. En el Cuadro FLUTTER 1.4.2 se describe el estilo según se aplica en Flutter para crear interfaces de usuario reactivas.



Flutter 1.4.2. Interfaces de usuario reactivas con Flutter

Se trata de un sistema pseudo-declarativo, en el cual, el desarrollador debe proporcionar un mapeo entre el estado de la aplicación y el de la interfaz. Cada vez que cambia el estado de la aplicación, el framework debe actualizar la interfaz en tiempo de ejecución. Está especialmente indicado para interfaces de usuario muy complejas, de forma que se ahorra mucho tiempo al desarrollador especificando qué aspecto debe tener la interfaz ante cualquier interacción del usuario y cambio del modelo (estado del sistema).

PREGUNTA 1.4.1. Programación reactiva y Manejador de eventos

¿Qué diferencia hay entre ambos estilos?

- Estilo de *Procesos Distribuidos*.- Dentro de éstas existen arquitecturas diversas según criterios diversos, como la topología que relaciona los procesos o el protocolo de comunicación entre procesos. Por ejemplo, atendiendo a la topología de la red podemos tener estilos arquitectónicos en anillo o en estrella.
 - Estilo *Peer-to-Peer (P2P)*.- Se refiere a todas las arquitecturas de componentes independientes en las que cada componente o nodo tiene las mismas capacidades y responsabilidades. Dos ejemplos en software de intercambio de archivos son eMule y BitTorrent.
 - Estilo *Cliente/Servidor*.- El estilo *Cliente/Servidor* es un tipo particular de arquitectura de procesos distribuidos de uso muy generalizado en el que al menos entran en juego dos nodos con responsabilidades y capacidades muy distintas: el cliente (pudiendo haber varios) y el servidor. El servidor representa a los procesos que dan servicio y el cliente a los procesos que reciben el servicio. El servidor no conoce a los clientes por adelantado, mientras que el cliente conoce al servidor y accede a él mediante llamada a procedimientos remota (remote procedure call, RPC). También se considera un subtipo del estilo *Sistema por Capas* con solo dos capas aunque realmente puede haber más capas dentro del servidor o del cliente. Dentro de este estilo, podemos encontrar al menos dos subtipos de arquitecturas:
 - *Arquitectura Orientada a Servicios* (del inglés *Services Oriented Architecture (SOA)*).- Arquitectura en la cual, los componentes están fuertemente desacoplados⁵⁸, de forma que unos proporcionan (servidores) y otros con-

⁵⁸Para ver su relación con un estilo de mayor acoplamiento del que proviene, la *Arquitectura basada en Componentes*, puede consultarse el trabajo de Helmut Petritsch Helmut Petritsch, *Service-Oriented Architecture (SOA) vs. Component Based Architecture*, informe técnico (2006), visitado 13 de febrero de 2023, http://petritsch.co.at/download/SOA_vs_component_based.pdf.

sumen (clientes) servicios, sin que éstos conozcan la implementación de los servidores (caja negra), logrando la interoperabilidad en la interacción entre máquinas, sistemas software y aplicaciones a través de la red. Además se trata de una arquitectura sin estado⁵⁹, que conectan entre sí para proveer nueva funcionalidad, por una interfaz bien definida.

Esta arquitectura se ha desarrollado especialmente por los servicios Web⁶⁰, «ya que poseen un conjunto de características que permiten cubrir todos los principios básicos de la orientación a servicios».⁶¹ En SOA, se consigue el desacoplamiento entre los agentes software que interactúan siguiendo dos principios:

1. Usan un conjunto reducido de interfaces simples accesibles por todos los proveedores y consumidores.
2. Usan mensajes descriptivos, entregados a través de las interfaces, más que instructivos, pues solo interesa saber qué servicio demandamos, no cómo se implementa internamente.

Un ejemplo de protocolo creado bajo esta arquitectura es el *protocolo SOAP (Simple Object Access Protocol)*.- Incluye (1) un protocolo con el mismo nombre (SOAP), basado en XML para pedir los servicios web mediante un protocolo de transporte (HTTP, SMTP, etc.); (2) un directorio donde publicar los servicios Web ofertados con la información necesaria para usarlos (UDDI: Universal Description, Siscovery and Integration); y (3) un lenguaje basado en XML para describir los servicios Web a publicar en un directorio UDDI: WSDL (Web Sevices Description Language).

A su vez, dentro del estilo arquitectónico SOA encontramos al menos dos subtipos arquitectónicos:

- ◊ *Arquitectura Basada en Recursos: REpresentational State Transfer (REST)*.- Se basa en el concepto de recurso: cualquier cosa con una URI (Uniform Resource Identifier). Permite distintos formatos de texto (plano, HTML, XML, JSON, etc.), aunque lo más frecuente es usar JSON. También podría usar el protocolo SOAP, basado en XML pero más complicado y requiriendo, por tanto, un mayor ancho de banda. Los servicios y los proveedores de servicios deben ser recursos. Aunque es un subtipo más de SOA, para algunos, tiene otras características que la hacen diferente.⁶² Las cinco características necesarias para tratarse de arquitectura

⁵⁹Cada petición del cliente al servidor debe contener toda la información necesaria para procesar la petición, sin poder hacer uso de ninguna información de contexto almacenada en el servidor. Por eso la información del estado de una sesión se almacena por completo en el cliente (cookies).

⁶⁰«Conjunto de protocolos, estándares y recomendaciones, definidos por la W3C (World Wide Web Consortium) y OASIS (Organization for the Advancement of Structured Information Standards).» Alberto Los Santos Aransay, *Revisión de los Servicios Web SOAP/REST: Características y Rendimiento*, informe técnico (Universidad de Vigo, 2009), http://www.albertolsa.com/wp-content/uploads/2009/07/mdsw-revision-de-los-servicios-web-soap_rest-alberto-los-santos.pdf.

⁶¹Los Santos Aransay.

⁶²Reynoso y Kicillof, *Estilos y patrones en la estrategia de arquitectura de Microsoft.*; Los Santos Aransay,

REST son:

1. Arquitectura cliente-servidor (como cualquier SOA).
2. Sin estado (como cualquier SOA).
3. Cacheable.- Para mejorar la eficiencia de la red, las respuestas deben ser capaces de ser etiquetadas como cacheables o no cacheables. La caché del cliente podrá usar una respuesta anterior si era cacheable.
4. Interfaz uniforme.- Esta característica simplifica la arquitectura. Para hacerla uniforme, REST aplica cuatro restricciones a sus interfaces, que se muestran en Tabla 1.3. Puede consultarse el Cuadro PARA PROFUNDIZAR 1.4.1 para más detalles sobre la forma más convencional de uso REST para garantizar la uniformidad de sus interfaces.
5. Sistema por capas.- Se permiten capas entre el cliente y el servidor, de forma que desde una capa solo se tiene acceso a las adyacentes. Por ejemplo, se pueden usar capas intermedias para mejorar la seguridad o el rendimiento (servidores proxy, caché, puertas de enlace, ...).
 - ◊ *Arquitectura de Computación en la Nube*.- Se trata de un tipo de *Sistema por Capas*, también orientado a servicio, como SOA, pero en el que los servicios finales consisten en recursos computacionales (almacenamiento y procesamiento), y los clientes (demandantes) no manejan directamente las peticiones a un servidor concreto sino que los proveedores les proporcionan lo que necesitan según los recursos disponibles en cada momento. Los servicios además no son horizontales, como en SOA básico, sino que se organizan por capas (v.g. de infraestructura, de plataforma y de aplicación) para proporcionar el servicio demandado por el cliente.

La figura 1.67 muestra la relación entre distintos superestilos y subestilos arquitectónicos de la arquitectura orientada a servicios (SOA).

- Estilos específicos de un dominio.- Como por ejemplo estilos concretos para la aviación, los videojuegos o los sistemas de gestión de vehículos. Al estar el domino restringido el estilo o patrón es más específico y permite generar código a partir de ella de forma automática o semi-automática.

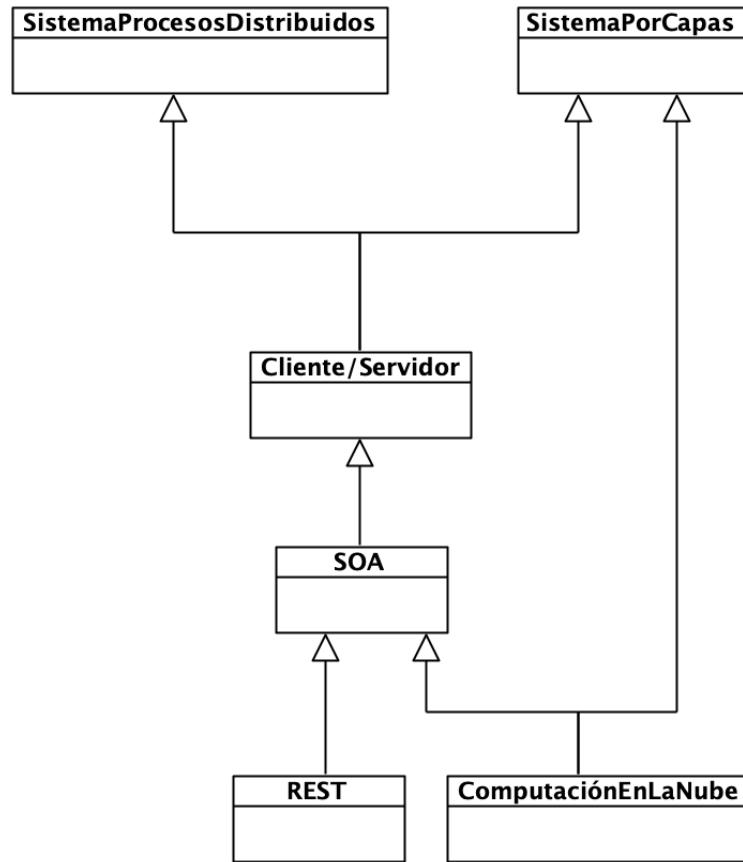


Figura 1.67: Relación de distintos superestilos y subestilos arquitectónicos de la arquitectura orientada a servicios (SOA).

Restricción	Descripción
URI	Identificación de los recursos
Representaciones	Manipulación de los recursos mediante representaciones, siendo una <i>representación</i> , el estado de un recurso en un momento concreto
Mensajes auto-descriptivos	El cliente tiene que entender el formato usado en las distintas representaciones de los recursos (lo que se llama <i>tipos de medios</i> , del inglés <i>media types</i>)
Hipermedia o hipertexto	Más genérico que usar navegadores y HTML, XML o JSON

Tabla 1.3: Restricciones adicionales de REST para hacer las interfaces uniformes.

PARA PROFUNDIZAR 1.4.1: REST e interfaz uniforme

De forma convencional se asocian los métodos recurso de REST con los métodos GET/PUT/POST/DELETE del protocolo HTTP, aunque lo único que establece REST es que se use una interfaz uniforme.

Así, las interfaces se construyen solo sobre HTTP, lo que las hace muy simples, con siete funciones definidas siguiendo el protocolo HTTP: GET, DELETE, HEAD, OPTION, POST, PUT, PATCH y TRACE, todas idempotentes^a, salvo POST. Las cuatro más usadas en muchas implementaciones REST (RESTFul) pueden verse en la Tabla 1.4. En la Figura 1.68 puede verse un ejemplo de uso.

Pero tampoco es necesario para implementar REST seguir esta convención. Se podría usar POST para actualizar un recurso en vez de PUT. Lo único que se exige es la uniformidad de la interfaz (que se haga de la misma manera con todos los recursos que ofrece el servicio Web).

También de forma convencional se suele usar XML y mucho más JSON como formatos de datos transmitidos. En un intento de hacer más estándar la Web, se recomienda usar los principios REST de forma más restrictiva, y por ello a menudo se identifica HTTP con REST. Como además, se eligen los cuatro métodos HTTP y se asocia cada uno de ellos a una de las cuatro operaciones (Create/Read/Update/Delete (CRUD) o CLAB en español) de persistencia de datos (bases de datos) en un sistema software, hay una triple asociación entre los conceptos CRUD, REST y HTTP, como se ve en la Tabla 1.5 con un ejemplo.

En la Tabla 1.6 se pone un ejemplo en una API de Ruby on Rails junto con las acciones del controlador que provoca cada ruta (URI). Como se ve para la operación *index*, el mismo recurso puede tener asociadas más de una URI.

^aUna función es idempotente cuando se produce el mismo efecto independientemente del número de veces que sea invocada.

Operación	Descripción
HTTP GET	Usado para obtener una representación de un recurso. Un consumidor lo utiliza para obtener una representación desde una URI. Los servicios ofrecidos a través de este interfaz no deben contraer ninguna obligación respecto a los consumidores
HTTP DELETE	Se usa para eliminar representaciones de un recurso. La segunda y siguientes veces que se hace, el recurso ya habrá sido borrado (Error 404) y el estado del recurso no cambia pues no existe ⁶³
HTTP POST	Usado para crear un recurso. NO es idempotente
HTTP PUT	Se usa para actualizar el estado de un recurso. Una vez actualizado, las siguientes repeticiones de esta operación, no provocarán ningún cambio en el recurso

Tabla 1.4: Las cuatro operaciones HTTP más habituales en REST.

CRUD	HTTP	REST
Create	POST	/api/project
Read	GET	/api/project/id
Update	PUT	/api/project/id
Delete	DELETE	/api/project/id

Tabla 1.5: Ejemplo de la relación entre CRUD, HTTP y REST sobre «project».

⁶³Una API que permita borrar el último objeto con la función DELETE: DELETE /item/last no cumple con la propiedad de idempotencia del método DELETE que exige HTTP y no sería una buena API REST. La solución para borrar de esta manera es usar el protocolo POST.

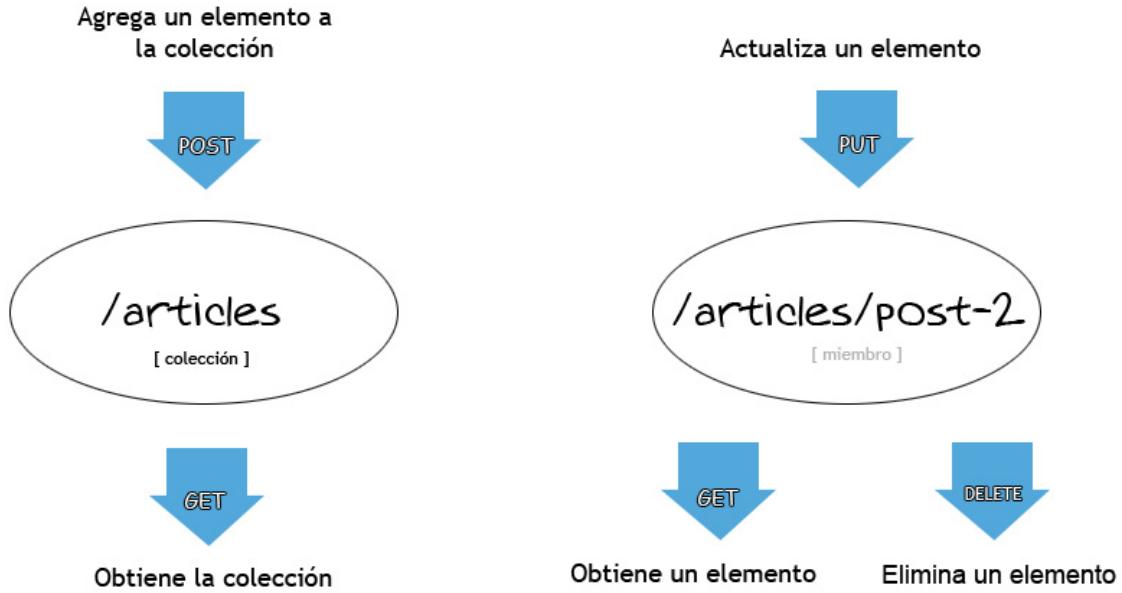


Figura 1.68: Ejemplo de uso de las funciones REST [Fuente: (Alberto Los Santos Aransay, *Revisión de los Servicios Web SOAP/REST: Características y Rendimiento*, informe técnico [Universidad de Vigo, 2009], http://www.albertolsa.com/wp-content/uploads/2009/07/mdsw-revision-de-los-servicios-web-soap_rest-alberto-los-santos.pdf)]

CRUD	HTTP	REST	Acción controlador
Create	POST	<code>/taller_rails/project</code>	<code>projects#create</code>
Read	GET	<code>/taller_rails/projects/id</code>	<code>projects#show</code>
Read	GET	<code>/taller_rails/projects</code>	<code>projects#index</code>
Read	GET	<code>/taller_rails</code>	<code>projects#index</code>
Read	GET	<code>/taller_rails/projects/new</code>	<code>projects#new</code>
Read	GET	<code>/taller_rails/projects/id/edit</code>	<code>projects#edit</code>
Update	PUT	<code>/taller_rails/projects/id</code>	<code>projects#update</code>
Update	PATCH	<code>/taller_rails/projects/id</code>	<code>projects#update</code>
Delete	DELETE	<code>/taller_rails/projectid</code>	<code>projects#destroy</code>

Tabla 1.6: Ejemplo de la relación entre CRUD, HTTP, REST y las operaciones del controlador en una aplicación realizada con Ruby on Rails: http://clados.ugr.es/taller_rails.

1.4.10. Combinación de estilos y frontera débil con patrones de diseño

En la realidad, los estilos se combinan de forma que en un solo sistema puede aplicarse más de uno. Se puede considerar que un componente implementa a su vez otro estilo y así sucesivamente, de forma que se relacionan entre ellos de forma jerárquica. Un ejemplo es el estilo *Tubería y Filtro*, como por ejemplo ocurre en las tuberías de Unix, que pueden programarse mediante la línea de comandos. Cada filtro puede implementar a su vez otro estilo arquitectónico.

También es posible que un mismo componente forme parte de más de un estilo, porque tenga conectores de varios estilos, como los estilos *Repositorio*, *Tubería y Filtro* y *Control de Procesos*. Así, la interfaz tendrá una parte específica para cada tipo de conector.

Por otro lado, la frontera entre estilo arquitectónico y patrón de diseño no está siempre clara, en especial cuando bajamos en el nivel de anidamiento entre estilos arquitectónicos en sistemas que implementan más de uno.

Acrónimos

CRUD Create/Read/Update/Delete

MVC Modelo-Vista-Controlador

OO Orientado a Objetos

REST REpresentational State Transfer

Bibliografía

Appleton, Brad. *Patterns and Software: Essential Concepts and Terminology*, 2000. <http://www.bradapp.com/docs/patterns-intro.html>.

Balachandran Pillai, Anand. *Software Architecture with Python*. Packt Publishing, 2017. <https://learning.oreilly.com/library/view/software-architecture-with/9781786468529/ch08s04.html>.

Bass, Len, Paul Clements y Rick Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2003.

Beck, Kent y Ward Cunningham. «Using Pattern Languages for Object Oriented Programs». En *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1987. <http://c2.com/doc/oopsla87.html>.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. <https://learning.oreilly.com/library/view/pattern-oriented-software-architecture/9781118725269/>.

Gamma, Erich, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1994. <https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/>.

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

Gamma, Erich, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software- CD*. USA: Addison-Wesley Longman Publishing Co., Inc., 1994.

Garfixia. *Pipe-And-Filter*. Visitado 4 de marzo de 2020. http://www.dossier-andreas.net/software_architecture/pipe_and_filter.html.

Los Santos Aransay, Alberto. *REST API Tutorial*. Visitado mayo de 2021. <https://restfulapi.net>.

- Los Santos Aransay, Alberto. *Revisión de los Servicios Web SOAP/REST: Características y Rendimiento*. Informe técnico. Universidad de Vigo, 2009. http://www.albertolsa.com/wp-content/uploads/2009/07/mdsw-revision-de-los-servicios-web-soap_rest-alberto-los-santos.pdf.
- Petritsch, Helmut. *Service-Oriented Architecture (SOA) vs. Component Based Architecture*. Informe técnico. 2006. Visitado 13 de febrero de 2023. http://petritsch.co.at/download/SOA_vs_component_based.pdf.
- Phillips, Greg, Rick Kazman, Mary Shaw y Florian Mattes. *Process Control Architectures*. Royal Military College of Canada, 1999. <https://www.slideshare.net/ahmad1957/process-control>.
- Reenskaug, Trygve. *A note on DynaBook requirements*. Informe técnico. Xerox PARC, 1979. <http://folk.uio.no/trygver/1979/sysreq/SysReq.pdf>.
- Reynoso, Carlos y Nicolás Kicillof. *Estilos y patrones en la estrategia de arquitectura de Microsoft*. Informe técnico. Universidad de Buenos Aires, 2004. <http://biblioteca.udgvirtual.udg.mx/jspui/handle/123456789/940>.
- Shaw, Mary y David Garlan. *Software Architecture*. New Jersey: Prentice Hall, 1996.
- Züllighoven, Heinz. *Object-Oriented Construction Handbook*. EE.UU.: Science Direct, 2005.

Tema 2. Mantenimiento y evolución del software

Desarrollo de Software

Curso 2022-2023

3º - Grado Ingeniería Informática

Dpto. Lenguajes y Sistemas Informáticos

ETSIIT

Universidad de Granada

12 de mayo de 2023



Tema 2. Mantenimiento y evolución del software

Contenidos

2.1. Evolución vs mantenimiento	6
2.2. Evolución software	6
2.3. Mantenimiento software	7
2.3.1. Especificidades de mantenimiento del software basado en componentes	9
2.4. Modelos y procesos de mantenimiento y evolución software	10
2.5. Estudio de impacto software	11
2.6. Reingeniería	14
2.6.1. Razones o necesidades por las que se decide hacer reingeniería	14
2.6.2. Conceptos de reingeniería	15
2.7. Software heredado (legacy software)	19
2.7.1. Métodos de migración	21
2.8. Refactorización y reestructuración	23
2.8.1. Identificar qué refactorizar	24
2.8.2. Cómo verificar la preservación del comportamiento (observable) del sistema	24
2.8.3. Refactorización básica y compleja	24
2.9. Comprensión de un programa	25
Acrónimos	27
Bibliografía	29

Tema 2. Mantenimiento y evolución del software

Contenidos

Mantenimiento y evolución del software

Este capítulo ha sido extraído principalmente del primer capítulo del libro de Priyadarshi Tripathy y Kshirasagar Naik titulado *Software Evolution and Maintenance*.¹ Para evitar repetidas referencias al mismo trabajo, sólo se citarán de forma explícita otros autores, entendiéndose que el texto no citado es una traducción literal o resumida del libro mencionado.

¹Priyadarshi Tripathy y Kshirasagar Naik, *Software Evolution and Maintenance* (EE.UU.: John Wiley, 2014), <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>.

2.1. Evolución vs mantenimiento

El concepto de *mantenimiento software* se refiere a corregir los errores del software que le impiden que realice las funcionalidades previstas en la entrega. Así, todas las actividades de soporte realizadas después de la entrega del software se incluyen en la categoría de mantenimiento. El mantenimiento de los sistemas de software significa principalmente corregir errores pero preservar sus funcionalidades. Las tareas de mantenimiento están muy planificadas, como por ejemplo la que se refiere a la corrección de errores. Además de las actividades planificadas, también se realizan actividades no planificadas. Por ejemplo, puede surgir un nuevo uso del sistema. En general, el mantenimiento no implica realizar cambios importantes en la arquitectura del sistema. En otras palabras, el mantenimiento significa mantener un sistema instalado funcionando sin cambios en su diseño.

El concepto de *evolución software* se refiere a un cambio continuo del software desde un estado menor, más simple o peor, a un estado más alto o mejor. Así, todas las actividades realizadas para efectuar cambios en los requisitos se consideran dentro de la categoría de evolución. La evolución de un sistema software significa ampliar un diseño que ya existe. Algunos ejemplos son la adición de funcionalidad, la mejora del rendimiento del sistema o el hacerlo ejecutable en un sistema operativo diferente. Conforme pasa el tiempo, las partes interesadas comprenden mejor lo que el sistema debe hacer y ponen manos a la obra para que se acerque al nuevo concepto que tienen del mismo. Por lo tanto, el sistema evoluciona de varias maneras, pero siempre es primero la persona humana la que empieza a “soñarlo” de manera distinta.

2.2. Evolución software

A continuación se exponen las primeras tres leyes de la evolución del software:²

- Ley de cambio continuo.- A menos que un sistema se modifique continuamente para satisfacer las necesidades emergentes de los usuarios, el sistema se vuelve cada vez menos útil.
- Ley de entropía/complejidad creciente.- A menos que se realice un trabajo adicional para reducir explícitamente la complejidad de un sistema, el sistema se volverá cada vez más complejo y menos estructurado debido a los cambios relacionados con el mantenimiento.
- Ley de crecimiento estadísticamente suave o autorregulación.- El proceso de evolución puede parecer aleatorio en vistas puntuales de tiempo y de espacio pero a lo largo del tiempo, se aprecia a nivel estadístico una autorregulación (ley de conservación) cíclica, con tendencias bien definidas a largo plazo. Así, las medidas del nivel de mantenimiento requerido por los distintos productos y procesos, que se producen durante la evolución, siguen una distribución cercana a la normal.

²L. A. Belady y M. M. Lehman, «A model of large program development», ed. por Springer, *IBM Systems Journal* 1, n.º 15 (1976): 225-252.

Se han dado hasta ocho leyes de evolución, pero algunas son bastante controvertidas. En concreto se considera que pueden variar según se estudie la evolución en sistemas software registrados o patentados y monolíticos, software académico, software industrial, software gratuito, software de código abierto, etc.

2.3. Mantenimiento software

E. Burton Swanson definió inicialmente tres categorías de actividades de mantenimiento software, a saber, *correctivo*, *adaptativo* y *perfectivo*. Esas definiciones se incorporaron posteriormente a la ingeniería del software estándar (procesos del ciclo de vida del software). Posteriormente se introdujo una cuarta categoría llamada mantenimiento *preventivo*, aunque algunos investigadores y desarrolladores la consideran como un subconjunto del mantenimiento perfectivo.

La clasificación de Swanson de las actividades de mantenimiento se basa en el criterio de la intención del ingeniero de mantenimiento, pues reflejan las distintas intenciones de realizar tareas de mantenimiento específicas en el sistema. Así, la intención de una actividad depende de las motivaciones para el cambio. En el cuadro 2.3.1 aparecen las definiciones de los cuatro tipos de mantenimiento, según la motivación para el cambio, tal y como han sido adoptados por el *Standard for Software Engineering–Software Maintenance, ISO/IEC 14764*.

DEFINICIÓN 2.3.1: Los cuatro tipos de mantenimiento según el *Standard for Software Engineering–Software Maintenance, ISO/IEC 14764*

Mantenimiento correctivo.- «El propósito del mantenimiento correctivo es corregir fallas: fallas de procesamiento y fallas de rendimiento. Un programa que produce una salida incorrecta es un ejemplo de falla de procesamiento. De manera similar, un programa que no puede cumplir con los requisitos en tiempo real es un ejemplo de falla de rendimiento. El proceso de mantenimiento correctivo incluye el aislamiento y corrección de elementos defectuosos en el software. El producto de software se repara para satisfacer los requisitos. Hay una variedad de situaciones que pueden describirse como mantenimiento correctivo, como corregir un programa que aborta o produce resultados incorrectos. Básicamente, el mantenimiento correctivo es un proceso reactivo, lo que significa que el mantenimiento correctivo se realiza después de detectar defectos en el sistema.»

Mantenimiento adaptativo.- «El propósito del mantenimiento adaptativo es permitir que el sistema se adapte a los cambios en su entorno de datos o entorno de procesamiento. Este proceso modifica el software para interactuar correctamente con un entorno cambiante o modificado. El mantenimiento adaptativo incluye cambios, adiciones, eliminaciones, modificaciones, extensiones y mejoras del sistema para satisfacer las necesidades cambiantes del entorno en el que debe operar el sistema. Algunos ejemplos genéricos son: (i) cambiar el sistema para admitir una nueva configuración de hardware; (ii) convertir el sistema de operación por lotes a operación en línea; y (iii) cambiar el sistema para que sea compatible con otras aplicaciones. Un ejemplo más concreto es: un software de aplicación en un teléfono inteligente puede mejorarse para soportar la comunicación basada en Wi-Fi además de su actual comunicación celular de Tercera Generación (3G).»

Mantenimiento perfectivo.- «El propósito del mantenimiento perfectivo es realizar una variedad de mejoras, a saber, la experiencia del usuario, la eficiencia del procesamiento y la mantenibilidad. Por ejemplo, los resultados del programa se pueden hacer más legibles para una mejor experiencia del usuario; el programa se puede modificar para hacerlo más rápido, aumentando así la eficiencia del procesamiento; y el programa se puede reestructurar para mejorar su legibilidad, aumentando así su mantenibilidad. En general, las actividades para el mantenimiento perfectivo incluyen la reestructuración del código, la creación y actualización de la documentación y el ajuste del sistema para mejorar el rendimiento. También se le llama “mantenimiento por el bien del mantenimiento” o “reingeniería”. Además, también debemos considerar mantenimiento perfectivo la adición de nueva funcionalidad en el sistema para responder a nuevas necesidades surgidas con el tiempo.

Mantenimiento preventivo.- «El propósito del mantenimiento preventivo es evitar que ocurran problemas al modificar los productos de software. Básicamente, uno debe mirar hacia adelante, identificar riesgos futuros y problemas desconocidos, y tomar acciones para que esos problemas no ocurran. Por ejemplo, los buenos estilos de programación pueden reducir el impacto del cambio, reduciendo así la cantidad de fallas. Por lo tanto, el programa se puede reestructurar para lograr buenos estilos para facilitar la comprensión posterior del programa. El mantenimiento preventivo se realiza muy a menudo en sistemas de software críticos para la seguridad y de alta disponibilidad. El concepto de “rejuvenecimiento del software” es una medida de mantenimiento preventivo para evitar, o al menos posponer, la ocurrencia de fallas debido a la ejecución continua del sistema de software.»

Otro criterio de clasificación de las modificaciones al software se basa en las actividades que son realizadas:

- Actividades para hacer correcciones (similar al mantenimiento correctivo de Swanson).- Si hay discrepancias entre el comportamiento esperado de un sistema y el comportamiento real, entonces se realizan algunas actividades para eliminar o reducir las discrepancias.
- Actividades para realizar mejoras.- Se realizan una serie de actividades para producir un cambio en el sistema, cambiando así el comportamiento o la implementación del sistema. Esta categoría de actividades se refina aún más en tres subcategorías:
 - mejoras que modifican los requisitos existentes;
 - mejoras que crean nuevos requisitos; y
 - mejoras que modifican la implementación sin cambiar los requisitos.

PREGUNTA 2.3.1. Evolución de software y tipos de mantenimiento

¿Qué relación encuentras entre el concepto de evolución de software del apartado anterior y los tipos de mantenimiento vistos ahora?

2.3.1. Especificidades de mantenimiento del software basado en componentes

No es lo mismo enfrentarse a la tarea de mantener un sistema con código desarrollado por completo a medida, *software a medida*, que enfrentarse a las tareas, más habituales en la actualidad, de desarrollar *software basado en componentes* (**CBS**, del inglés *Component Based Software system*) de fuentes heterogéneas, en el que los distintos componentes provienen de distintas fuentes de *software comercial listo para usar* (**COTS**, de inglés *Commercial Off-the Shelf*) que se proporciona como componentes reutilizables, además de fuentes de desarrollo propias y de posiblemente código abierto. Las características específicas más importante del mantenimiento de CBSs de fuentes heterogéneas, incluyendo diversos componentes COTS son:

- Mantenimientos divididos.- Cada producto COTS lo mantiene el equipo de mantenimiento de la empresa que lo proporciona, que es distinto al equipo de la empresa que lo aplica en su propio software. Aunque los primeros pueden dar soporte a los segundos, el tener objetivos comerciales distintos puede añadir ciertas complicaciones.
- Habilidades específicas.- El que trabaja en un equipo de mantenimiento de un software basado en componentes, ve cada componente como una caja negra y se especializa más en la integración de los mismos.

- Comunidad de usuarios mayor: confianza vs control.- Siempre tiene ventajas que sean más los usuarios que utilizan un producto para garantizar un mejor mantenimiento. Aunque no podemos controlar los cambios concretos en un producto desarrollado por terceras partes, hay que confiar en que el mayor número de usuarios a la larga dirigirá la evolución del componente hacia un lugar más provechoso.
- Desplazamiento de los costes.- Generalmente es más aconsejable usar software que ya está preparado que desarrollar el nuestro propio, si se ajusta bien a nuestras necesidades. Sin embargo, al ahorro en el desarrollo y mantenimiento de software propio hay que descontarle los gastos de adquisición (licencia) y de mantenimiento externos y los que se deriven de cambios importantes futuros que pudieran afectar al resto de los componentes de nuestro propio sistema.
- Planificación más difícil.- No se puede predecir cuándo se harán los cambios por las distintas empresas que desarrollan COTS usados por nuestro sistema.

2.4. Modelos y procesos de mantenimiento y evolución software

Se han propuesto muchos modelos que explican la labor de mantenimiento software. En el clásico modelo en cascada del ciclo de vida software, consiste en una única fase final, que empieza una vez que el software está en explotación.

Uno de estos modelos, explica el mantenimiento a su vez como un proceso en fases, con las siguientes etapas:

- Desarrollo inicial.- Aún no ha comenzado el mantenimiento, y el sistema está constantemente siendo modificado hasta que alcanza el estado de estabilidad suficiente para lanzarlo y empezar a usarlo.
- Evolución.- Nada más lanzado es fácil hacer cambios simples. Los más complejos llevan más costos y riesgos, pero son asumibles en su gran mayoría. Los desarrolladores pueden encargarse del mantenimiento aunque se dediquen a otras tareas, pues tienen el sistema fresco y les cuesta poco mantenerlo. Un ejemplo de modelo de mantenimiento en esta fase es el llamado modelo de *cambio en mini-ciclo*, del inglés *change mini-cycle*, que considera que hay, a su vez, que realizar cinco grandes fases para atender a una **Solicitud de Cambio (SC)**, **Change Request (CR)** en inglés. El primer paso sería la propia **SC**, y después seguirían el análisis y planificación del cambio³ (2), la implementación del cambio⁴ (3), la verificación y validación del cambio (4) y la documentación del cambio (5). La figura 2.1 muestra un diagrama con este modelo.

³Aquí se debe incluir un estudio de impacto (ver sección 2.5).

⁴Aquí se puede incluir refactorización y reestructuración del código si la **SC** corresponde a mantenimiento preventivo o perfectivo (ver sección 2.8).

- Servicio de mantenimiento.- Poco a poco sus desarrolladores ya no están disponibles y el nuevo equipo de desarrollo se dedica a hacer sobre todo mantenimiento correctivo (arreglar errores). Ya no se invierte en cambios mayores y cualquier cambio es mucho más costoso.
- Retirada gradual (“phaseout”).- En algún momento el servicio de mantenimiento mínimo se hace demasiado caro o bien aparecen soluciones mejores. Se tiene que planificar el cambio al nuevo sistema, incluyendo migración de datos y uso de patrones fachada o envoltorios. Cuando el nuevo sistema está completamente probado, el antiguo termina su servicio (a veces pueden funcionar durante cierto tiempo en paralelo). En esta fase, en la que el software antiguo aún se está usando pero ya se está desarrollando uno nuevo, a la parte del antiguo que se pretende “reutilizar” y formará parte del nuevo de alguna manera, se le suele denominar *software heredado* (en inglés, *legacy software*).

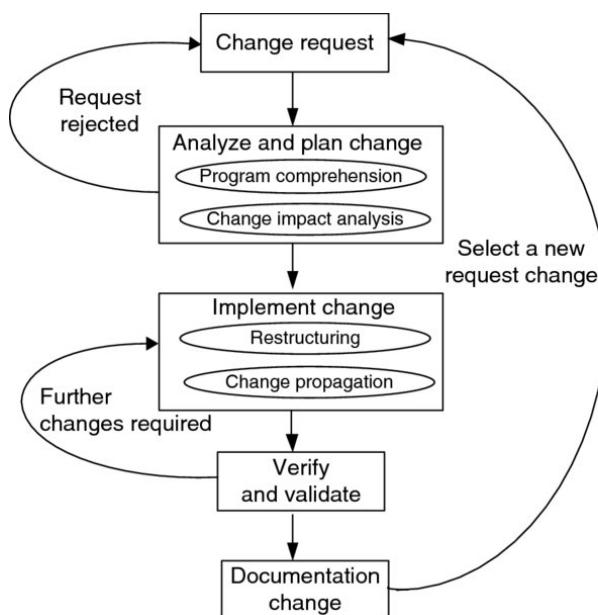


Figura 2.1: Modelo de cambio en mini-ciclo. [Fuente: (Priyadarshi Tripathy y Kshirasagar Naik, *Software Evolution and Maintenance* [EE.UU.: Johh Wiley, 2014], figura 3.8, <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>)]

2.5. Estudio de impacto software

El *análisis o estudio de impacto* es una tarea que pretende examinar los efectos potenciales que puede tener en las distintas partes del sistema la realización de una tarea concreta de mantenimiento. Se suelen usar dos tecnologías distintas para describir dicho impacto:

- *Análisis de trazabilidad*.- Pretende hacer un rastreo de todas las partes posibles afectadas o *artefactos de alto nivel* (diseño, código, casos de prueba, etc.) usando un

modelo de asociación entre los artefactos a modificar y los que pueden verse afectados. El cuadro 2.5.1 muestra un ejemplo de metodología para realizar un análisis básico, adaptado para el estilo [Orientado a Objetos \(OO\)](#).

- Análisis de dependencias.- Se describen los cambios a nivel de dependencias semánticas entre las distintas entidades del producto software mediante su identificación con las dependencias sintácticas que se extraen del código.

EJEMPLO 2.5.1. Análisis de trazabilidad en pequeños sistemas

Ante una **SC** en el mantenimiento, especificado, por ejemplo, como un nuevo requisito funcional en un mantenimiento perfectivo, el análisis de trazabilidad es un proceso iterativo, que identifica y actualiza en cada paso desde la especificación del requisito hasta su implementación final, el impacto que el cambio tendrá en el sistema, reevaluando la lista de posibles requisitos, diagramas de clases, métodos e implementación de los mismos (también documentación) que podrían verse afectados, y que se conoce como el *Conjunto de Impacto* o, en inglés, **IS**. Un primer boceto de este análisis consiste en identificar el llamado *Conjunto de Impacto Inicial* o, en inglés, **SIS**. Para ello, a partir de conceptos del sistema relacionados con la **SC** (que se identifican mediante sustantivos clave en la especificación del nuevo requisito funcional) se creará un grafo con cuatro grandes silos (nodos-contenedores), que serán, ordenados de izquierda a derecha, y según el ciclo de vida software: requerimientos, diseño (clases, en **OO**), código (métodos) y pruebas. La trazabilidad horizontal, es decir, cómo afectan los productos de una fase en la siguiente, se representará con arcos dirigidos (de la causa al efecto o producto en el que impacta) con líneas discontinuas. La trazabilidad vertical, es decir, cómo afectan/impactan productos dentro de una misma fase, se representará mediante arcos dirigidos (de la causa al efecto o producto en el que impacta) con líneas continuas. Dentro de cada silo (nodo-contenedor), se representarán los productos afectados directamente por productos de la fase anterior como pequeños nodos circulares en gris claro y los afectados indirectamente (por productos de la misma fase), de la misma forma pero en gris oscuro. Un ejemplo de conceptos, es decir, sustantivos clave o importantes, puede verse en esta descripción de un requisito funcional: «Añade al cajero automático un nuevo tipo de pago, mediante una tarjeta de débito expedida por el banco Chautauqua». Los conceptos que se pueden identificar mediante sustantivos clave podrían ser: ‘tarjeta débito’, ‘pago’. Cuando se usa **OO**, los conceptos serán implementados como clases y, algunos menos importantes, como simples atributos. En arquitecturas no **OO**, o en arquitecturas **OO** que hacen una transformación de los conceptos en clases de manera incorrecta (mal uso de la herencia, de la asociación, de las clases parametrizadas y de la reusabilidad en general) se puede hacer necesario buscar si los conceptos identificados en los requisitos, ya aparecían no solo en otros requisitos o clases sino en el código fuente mediante búsqueda de texto con herramientas como *grep* de linux o similares. Pero si se trata de una arquitectura **Orientado a Objetos (OO)** correctamente implementada, podemos movernos de izquierda a derecha del grafo pasando de los requisitos a las clases en el diagrama de clases, y de éstas a los métodos y de éstos a las pruebas. La figura 2.2 muestra un ejemplo de grafo de trazabilidad donde se ha añadido/cambiado el requisito funcional nº 4.

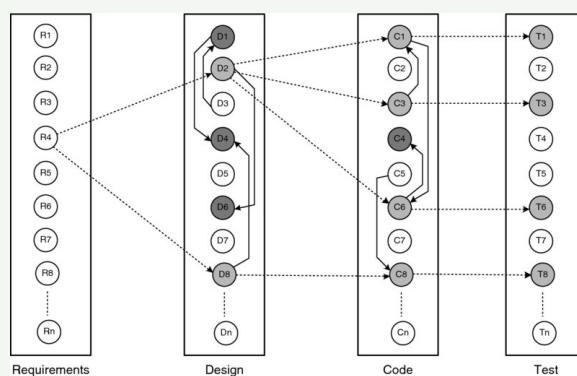


Figura 2.2: Modelo conceptual del estándar IEEE P1471. [Fuente: (Priyadarshi Tripathy y Kshirasagar Naik, *Software Evolution and Maintenance* [EE.UU.: John Wiley, 2014], figura 6.4, <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>)]

Un buen estudio de impacto debe incluir también un *análisis de los efectos de propagación en cadena* (en inglés, *ripple effect analysis*) que puede llevar un cambio. Se trata de ver el producto software como un todo evolutivo, un sistema, de forma que para cada nuevo cambio se mide:

- El cambio (aumento o disminución) de la complejidad del sistema con respecto a la versión anterior.
- Las diferencias en niveles de complejidad de las distintas partes del sistema.
- El efecto que tiene un nuevo módulo para la complejidad total del sistema.

2.6. Reingeniería

La *reingeniería* se define como la transformación de un sistema simple en otro mejor. Una forma alternativa de ver la evolución software y, en general, el ciclo de vida del software, es considerar un modelo en un nivel de abstracción más elevado en el que no se describa el proceso que sufre cada producto software sino el de un sistema completo que va evolucionando a través de distintos productos software que se suceden en el tiempo, de forma que cada producto parte del anterior, y no hay límite en la evolución. En este sentido, la evolución software puede verse como un proceso iterativo de reingeniería. En el análisis de efectos de propagación en cadena visto en el apartado anterior, se pone el énfasis en considerar el producto software como sistema evolutivo. Aún así, es un sistema que nace y muere. La reingeniería no pone la vista en el producto software concreto con su ciclo de vida, sino en un proceso a un nivel más alto de abstracción, formado por una cadena de productos software, en la que cada producto antes de morir ha permitido el nacimiento de otro nuevo, tratándose así de un proceso en mejora continua.

Una definición más detallada es considerar la reingeniería como el examen y análisis de un software existente (especificación, diseño, implementación y documentación), reestructurándolo y concibiéndolo de otra forma a partir de la cual se vuelva a implementar, mejorando la funcionalidad y los atributos de calidad del sistema, tales como capacidad de evolución, rendimiento, reusabilidad, eficiencia, portabilidad, etc. Se trata de pasar de un “mal” sistema a uno “bueno”, aunque puede haber algunos riesgos: (1) que la funcionalidad anterior no se mantenga; (2) que la calidad sea inferior; y (3) que los beneficios no se consigan en el tiempo esperados.

2.6.1. Razones o necesidades por las que se decide hacer reingeniería

Para llevarla a cabo, al menos debe surgir una de las siguientes necesidades:

- Mejorar la mantenibilidad.- Por la segunda ley de Lehman (*complejidad incremental*), el mantenimiento de un sistema aumenta con el tiempo hasta hacerse demasiado difícil y costoso. En algún momento habrá que hacer uno nuevo con interfaces más explícitas

y módulos funcionales más relevantes, actualizando además toda la documentación interna y externa del sistema.

- Migrar a una nueva tecnología.- Por la primera ley de Lehman (*cambio continuo*), los sistemas están en continua adaptación a su entorno siendo cada vez menos satisfactorios. La rapidez con la que avanza la tecnología hace que el software caduque más rápidamente. Además el software antiguo es más difícil de ser mantenido por las empresas, siendo cada vez más incompatible y caro de mantener. También los empleados van cambiando a las nuevas tecnologías, habiendo cada vez menos empleados que sean capaces de mantener sistemas antiguos. Así, muchas empresas con software operativo y útil se ven forzadas a migrarlos a plataformas de ejecución más modernas que incluyen nuevo hardware, sistema operativo y/o lenguaje.
- Mejorar la calidad.- Según la séptima ley de Lehman (*descenso de calidad*), las partes interesadas en un software perciben una bajada de calidad en los sistemas que no se mantienen de forma rigurosa ni se adaptan al entorno. Cada cambio produce efectos en cadena, a veces causando más problemas que mejoras hasta que la fiabilidad del software se hace insostenible. En ese momento se hace necesaria la reingeniería para conseguir aumentar la fiabilidad del mismo.
- Prepararse para una mejora de la funcionalidad.- La sexta ley de Lehman (*crecimiento continuo*) afirma que la funcionalidad del software debe estar constantemente ampliándose para mantener la satisfacción del usuario con ese software a lo largo de su tiempo de vida. Esta ley refleja el conjunto ilimitado de posibles mejoras en su funcionalidad. A menudo la reingeniería más que buscar añadir funcionalidad busca mejorar la facilidad de añadir funcionalidad en el futuro, por ejemplo cambiando el estilo arquitectónico o el paradigma de programación.

2.6.2. Conceptos de reingeniería

DEFINICIÓN 2.6.1: Ingeniería directa

Se trata de pasar del nivel más alto de abstracción en la representación de un sistema, donde solo se muestran las características más relevantes de un sistema escondiendo los detalles (principio de abstracción), al nivel más bajo del mismo, con el máximo detalle de sus distintos aspectos (principio de refinamiento).

Los pasos o actividades recorridas en la ingeniería directa empiezan por la formulación conceptual del sistema para identificar los requisitos y terminan con la implementación del diseño.

DEFINICIÓN 2.6.2: Ingeniería inversa

Se trata del movimiento contrario por el que se pasa de los niveles más altos de detalle a los niveles más altos de abstracción.

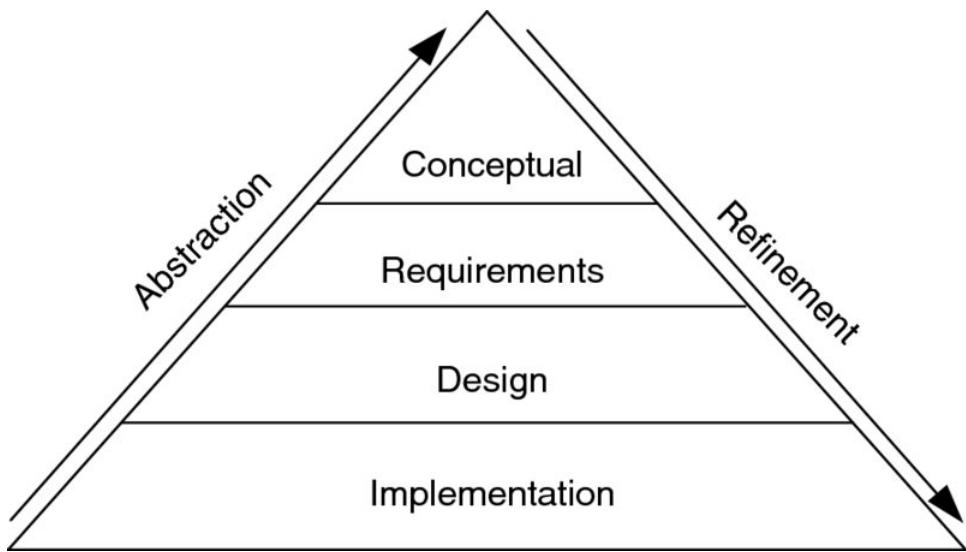


Figura 2.3: Niveles de abstracción y refinamiento. [Fuente: (Priyadarshi Tripathy y Kshirsagar Naik, *Software Evolution and Maintenance* [EE.UU.: John Wiley, 2014], cap. 4, <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>].

Los pasos o actividades de la ingeniería inversa empiezan por el análisis del software para entender sus componentes y la relación entre los mismos y terminan por la representación del sistema en un nivel más alto de abstracción o de una forma distinta.

Algunos ejemplos de ingeniería inversa son la descompilación (el código se traduce en un programa de alto nivel), la extracción arquitectónica (se deriva el diseño del programa a partir de su código), la generación de documentación (se produce información a partir del código fuente para comprender mejor el programa) y la visualización software (se dibujan algunos aspectos del programa con alguna forma de abstracción).

La figura 2.3 muestra la relación entre abstracción, refinamiento y cuatro niveles distintos de abstracción/refinamiento de un sistema software: (1) conceptual (*¿por qué* ese software?), (2) de requerimientos (*¿qué* hace el software?), (3) de diseño (*¿cómo* hace para conseguir esa funcionalidad?) y (4) de implementación (*¿cómo exactamente* es implementado el sistema?).

Junto a los principios de abstracción y refinamiento, en la reingeniería también interviene el principio de *alteración*, por el cual se hacen cambios en la representación de un sistema, sin cambiar el nivel de abstracción.

Un caso específico de alteración que no cambia el comportamiento externo del sistema es la *reestructuración*.

En el proceso de reingeniería se da la secuencia de abstracción->alteración->refinamiento. La figura 2.4 muestra esta secuencia.

La reingeniería ha sido definida de forma clásica con la siguiente ecuación:

$$\text{Reingeniería} = \text{ingeniería inversa} + \Delta + \text{ingeniería directa}.$$

Veamos los tres componentes de la parte derecha de la ecuación:

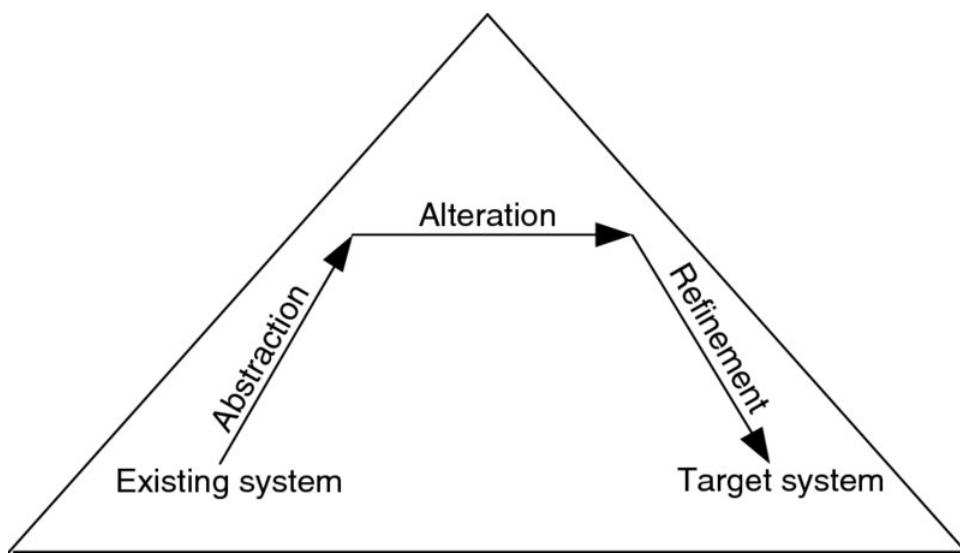


Figura 2.4: Bases conceptuales en el proceso de reingeniería. [Fuente: (Priyadarshi Tripathy y Kshirasagar Naik, *Software Evolution and Maintenance* [EE.UU.: Johh Wiley, 2014], cap. 4, <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>)].

- La *ingeniería inversa* consiste en la re-elaboración del modelo que representa al sistema actual de una forma más abstracta y fácil de entender. El punto de partida es el código actual y el resultado es una nueva DA o diseño de alto nivel del producto actual. No se trata todavía en esta fase de cambiar nada, sino de ver en profundidad, de examinar, el sistema actual.
- El segundo componente de la parte derecha de la ecuación, la *alteración* representa a la actividad de decidir los cambios que se harán para producir un nuevo producto software a partir del examen del producto actual. Se trata del proceso de especificación de requisitos y elaboración de una DA, pero partiendo del producto anterior.
- La *ingeniería directa* consiste en el desarrollo del nuevo producto software a partir de la nueva DA.

La Figura 2.5 muestra los distintos niveles de alteración que se corresponden con los distintos niveles de abstracción/refinamiento del código: repensar (cuando se cambia el sistema a nivel conceptual), re-especificar (cuando se cambia a nivel de requerimientos), rediseñar (cuando se cambia a nivel de diseño) y recodificación (cuando se cambia a nivel de implementación).

En la figura 2.6 puede verse un modelo similar, llamado modelo de herreradura, en el proceso de reingeniería. Obsérvese que este modelo reduce a tres los niveles de abstracción (arquitectónica, funcional y de código, de mayor a menor nivel). Bajo este modelo, los patrones y estilos de diseño afectan al nivel superior, el arquitectónico, que trabaja con los conceptos del sistema.

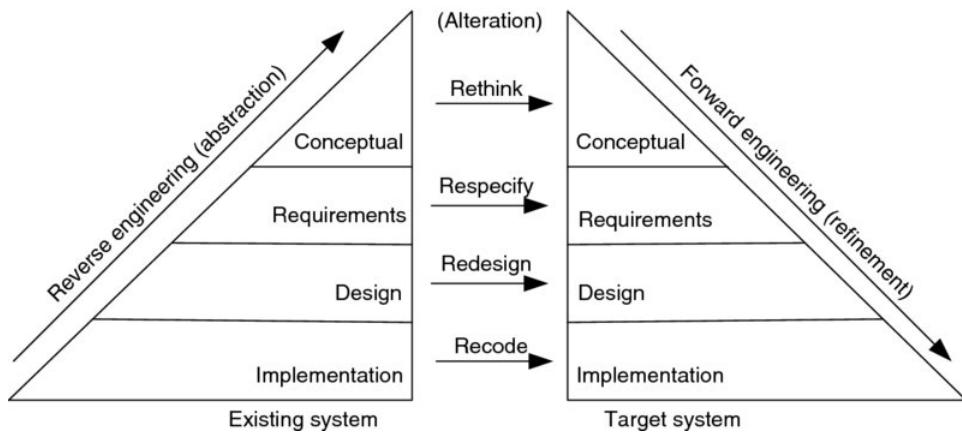


Figura 2.5: Bases conceptuales en el proceso de reingeniería. [Fuente: (Priyadarshi Tripathy y Kshirasagar Naik, *Software Evolution and Maintenance* [EE.UU.: Johh Wiley, 2014], cap. 4, <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>)].

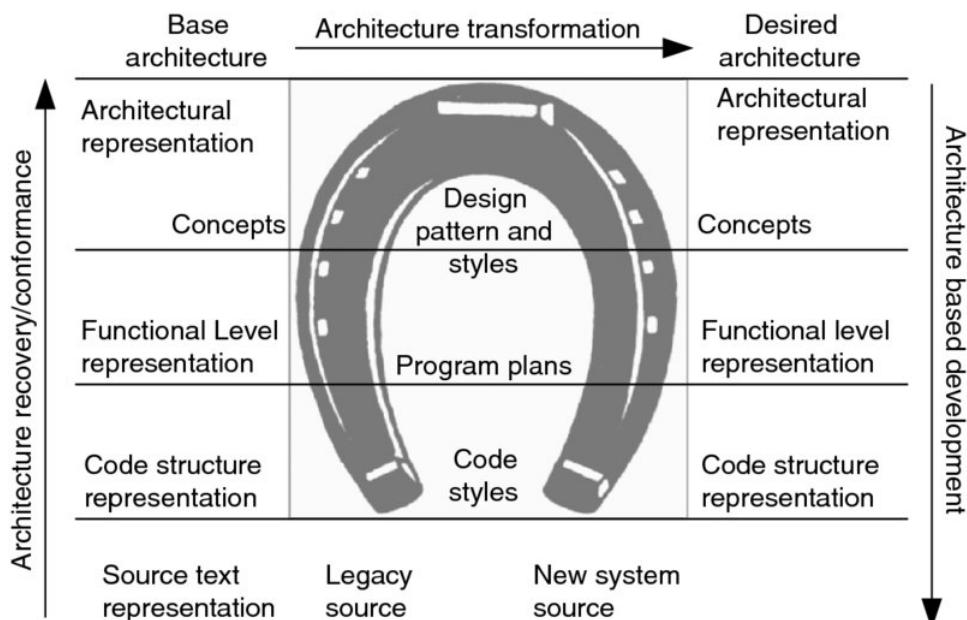


Figura 2.6: Bases conceptuales en el proceso de reingeniería. [Fuente: (Priyadarshi Tripathy y Kshirasagar Naik, *Software Evolution and Maintenance* [EE.UU.: Johh Wiley, 2014], cap. 4, <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>)].

2.7. Software heredado (legacy software)

A menudo las empresas tienen sistemas software que realizan funciones vitales pero que ya no se pueden mantener con facilidad por ser muy antiguos, escritos en lenguajes de programación obsoletos, mal documentados, con una mala gestión de los datos, con una estructura degradada después de sufrir muchas modificaciones, muy pocas personas tienen experiencia para hacer mínimos cambios, etc.

DEFINICIÓN 2.7.1: Software heredado

Sistema software casi imposible de mantener pero cuya funcionalidad es necesario mantener.

Si la empresa no puede prescindir de él, descarta seguir su mantenimiento y también descarta empezar desde cero, se presentan solo dos operaciones posibles:

- Envoltura (wrap).- Es una técnica de caja negra: se construye una capa exterior con una apariencia moderna y mejorada y se diseñan las interfaces para interaccionar con él, manteniendo ocultas la complejidad de sus interfaces, datos, módulos, etc. Se trata de una alternativa al mantenimiento. La figura 2.7 muestra un diagrama de un sistema envoltura. El *envoltorio (wrapper)* es un programa que recibe los mensajes desde el programa cliente y transforma la entrada en una representación que puede enviar al objetivo –el sistema heredado–. El *envoltorio* también intercepta la salida del objetivo, la transforma para hacerla inteligible al cliente y se la envía. Tiene para ellos los siguientes elementos (ver figura 2.7):
 - Interfaz externa.- Generalmente se usa paso de mensajes y los datos del mensaje son cadenas ASCII.
 - Interfaz interna.- Debe especificarse en el lenguaje del objetivo (el software heredado).
 - Gestor de mensajes.- Usa buffers de entrada y salida de mensajes para almacenar los datos que necesitan esperar dadas las distintas velocidades de procesamiento del cliente y el objetivo.
 - Convertidor de interfaces.- Se encarga de cambiar los parámetros necesarios para adaptarlos a cada una de las interfaces.
 - Emulador E/S.- Es el módulo que intercepta las entradas al objetivo y las salidas del mismo y pone la información en el buffer correspondiente.
- Migración.- Se trata de realizar reingeniería, de forma que el sistema sea exportado a una plataforma moderna, manteniendo la mayor parte de su funcionalidad y provocando los mínimos cambios en el modelo de negocio. Es más barato y rápido que empezar desde cero, pues se reutilizan partes importantes del mismo.

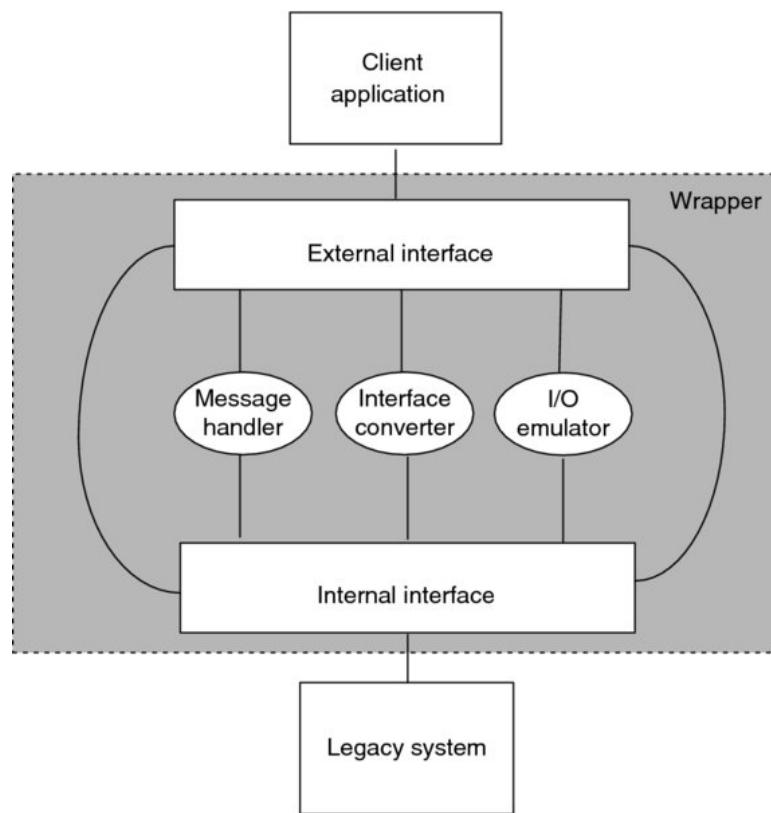


Figura 2.7: Módulos de un entorno envoltorio. [Fuente: (Priyadarshi Tripathy y Kshirasagar Naik, *Software Evolution and Maintenance* [EE.UU.: Johh Wiley, 2014], cap. 5, <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>)].

2.7.1. Métodos de migración

Existen diversos enfoques para llevar a cabo la migración. La elección del enfoque a aplicar dependerá del tipo concreto de software heredado, pues los distintos métodos presentan distintos niveles de complejidad, tamaño y riesgo de fallo al realizar la migración. Uno de los factores más decisivos que distinguen a los distintos métodos son las distintas formas de llevar a cabo la migración de los datos. Veremos tres de ellos:

Migración directa (forward migration) o Base de datos primero (database first)
Primero se migran los datos a un Sistema de Gestión de Bases de Datos (**SGBD**) más moderno y después de forma gradual el software y sus interfaces. Mientras concluye el segundo paso, conviven ambos sistemas mediante una *pasarela directa* (*forward gateway*) que media entre los distintos componentes software (ver figura 2.8).

Migración inversa (reverse migration) o Base de datos después (database last)
Los programas se van migrando de forma incremental pero permanece la base de datos del sistema heredado en su plataforma original. La interacción entre ambos sistemas de información durante el tiempo de la migración se hace mediante una *pasarela inversa* (ver figura 2.9).

Base de datos compuesta Combina los dos enfoques anteriores (ver figura 2.10): Los programas van migrando de forma incremental y en ese período en el que se usan los dos sistemas conviven también las dos bases de datos, mediante un sistema compuesto que usa una pasarela directa y otra inversa. Los datos pueden estar duplicados en las dos bases de datos y se usan *coordinadores de las transacciones* para asegurar su integridad. Estos coordinadores interceptan las peticiones de actualización de las bases de datos por parte del software heredado y nuevo, asegurándose de que los datos duplicados se actualicen en ambas bases de datos.

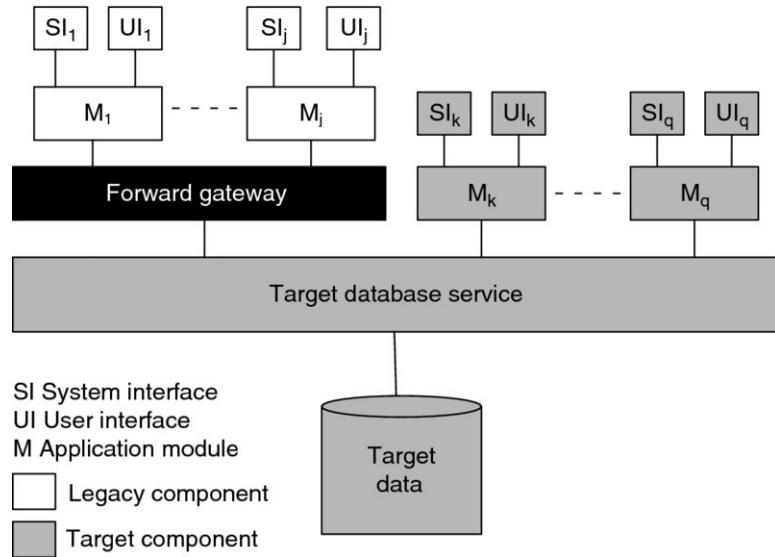


Figura 2.8: Enfoque *Base de datos primero*. [Fuente: (Priyadarshi Tripathy y Kshirasagar Naik, *Software Evolution and Maintenance* [EE.UU.: Johh Wiley, 2014], cap. 5, <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>)].

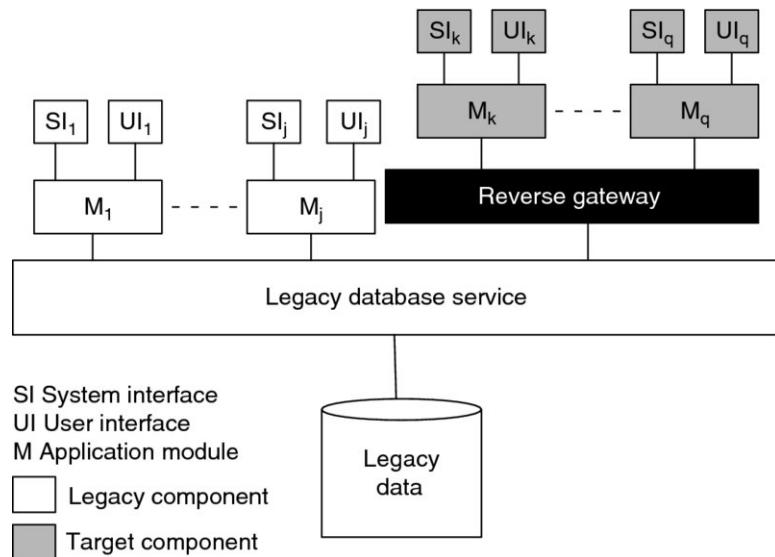


Figura 2.9: Enfoque *Base de datos después*. [Fuente: (Priyadarshi Tripathy y Kshirasagar Naik, *Software Evolution and Maintenance* [EE.UU.: Johh Wiley, 2014], cap. 5, <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>)].

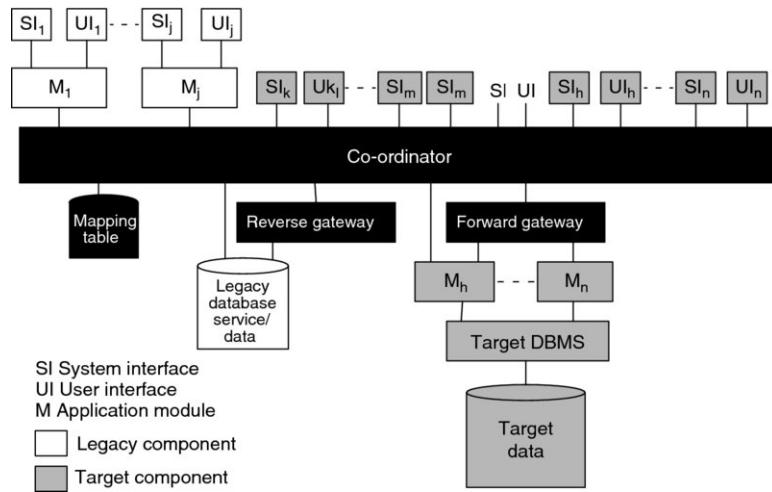


Figura 2.10: Enfoque *Base de datos compuesta*. [Fuente: (Priyadarshi Tripathy y Kshirasagar Naik, *Software Evolution and Maintenance* [EE.UU.: Johh Wiley, 2014], cap. 5, <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>)].

2.8. Refactorización y reestructuración

Reestructurar significa realizar cambios en la estructura del software para mejorar su calidad interna, es decir, para hacerlo más fácil de comprender y mantener, sin cambiar el comportamiento observable del sistema. En lenguajes no OO, la reestructuración se limita al nivel de una función o un bloque de código. En sistemas OO, con lenguajes mucho más ricos que usan interfaces, ligadura dinámica, herencia, sobreescritura, polimorfismo, etc. la reestructuración es más compleja y se la conoce con el nombre de *refactorización*.

La refactorización se logra mediante la eliminación del código duplicado, la simplificación del código y el movimiento del código a una clase diferente, entre otros. Sin una refactorización continua, la estructura interna del software se hará cada vez más difícil de comprender, debido al mantenimiento periódico. Por lo tanto, la refactorización regular permite mantener una buena estructura. En una metodología de software ágil, como eXtreme Programming (XP), la refactorización se aplica continuamente con los siguientes objetivos:

- proporcionar estabilidad en la arquitectura,
- proporcionar legibilidad al código, y
- facilitar el mantenimiento perfectivo y la evolución, flexibilizando las tareas de integración de nuevas funcionalidades en el sistema.

La refactorización no añade nuevos requisitos al sistema existente. Otro aspecto de la refactorización es mejorar la estructura interna del sistema. También se puede aplicar reestructuración para transformar código heredado y migrarlo a otro lenguaje de programación. Así, la reestructuración y refactorización se pueden utilizar para rediseñar sistemas software.

2.8.1. Identificar qué refactorizar

DEFINICIÓN 2.8.1: Hediondez del código (code smell)

Cualquier síntoma en el código fuente del software que puede indicar la presencia de un problema más serio.

Aunque un código que “huele” no implica errores, sí que tiene más posibilidades de errores en cambios futuros o mayor dificultad para hacer cambios. Algunos ejemplos de hediondez son:

- Código duplicado.- Fuente de errores futuros porque implica doble mantenimiento.
- Larga lista de parámetros.- Los posibles errores vienen de la facilidad para cambiar el orden en las llamadas sin darnos cuenta.
- Métodos grandes.- Difícil de entender, mantener y validar.
- Clases grandes.- Por ejemplo, más de 8 métodos o más de 15 variables son difíciles de mantener.
- Cadena de mensajes.- Por ejemplo: `student.getID().getRecord().getGrade(course)`. Significa que falta algún método o función que permita hacer ese encadenamiento.

2.8.2. Cómo verificar la preservación del comportamiento (observable) del sistema

Se verifica asegurando que todas las pruebas pasadas antes de refactorizar se puedan pasar después de refactorizar. Se trata de utilizar las llamadas *pruebas de regresión*.

2.8.3. Refactorización básica y compleja

La refactorización incluye una serie concreta de actividades básicas, las cuales se pueden combinar para formar refactorizaciones complejas. Las transformaciones básicas (código OO) son:

1. agregar una clase, método o atributo;
2. renombrar una clase, método o atributo;
3. mover un atributo o método hacia arriba o hacia abajo en la jerarquía;
4. eliminar una clase, método o atributo; y
5. extraer fragmentos de código en métodos separados.

2.9. Comprensión de un programa

Para poder formar parte de un equipo de mantenimiento en un software desconocido para nosotros, debemos dedicar todo el tiempo necesario a la comprensión del programa. Se trata de una tarea clave, que puede representar alrededor del 50 % del esfuerzo total dedicado al mantenimiento de un producto software y que repercute directamente en los costes de mantenimiento. En términos de actividades concretas, la comprensión del programa implica la construcción de modelos mentales de un sistema subyacente en diferentes niveles de abstracción, que varían desde modelos de bajo nivel del código hasta modelos de muy alto nivel del dominio de la aplicación subyacente. Los científicos cognitivos han estudiado modelos mentales para comprender cómo los seres humanos conocen, perciben, toman decisiones y construyen comportamientos en un mundo real. En el dominio de la comprensión de software, un modelo mental describe la representación mental que tiene un programador de un programa concreto.

Un paso clave en el desarrollo de modelos mentales es generar hipótesis o conjeturas e investigar su validez. Es algo que hacemos generalmente de forma implícita cuando queremos entender algo en profundidad. En el caso de un software que debemos mantener, el nivel de conocimiento debe ser muy alto pues las consecuencias de entender algo mal y hacer cambios pueden ser muy dañinas.

De esta manera, las hipótesis se deben formular de forma explícita, como parte de la importantísima tarea concreta de comprensión del programa. Las hipótesis son una forma de que un programador entienda el código de manera incremental. Después de comprender el código, el programador formula una hipótesis y la verifica leyendo el código. La verificación de la hipótesis da como resultado aceptar la hipótesis o rechazarla. A veces, una hipótesis puede no ser completamente correcta debido a la comprensión incompleta del código por parte del programador. Al formular continuamente nuevas hipótesis y verificarlas, el programador comprende cada vez más código y cada vez más detalles.

Se pueden aplicar varias estrategias para llegar a hipótesis significativas, como las estrategias de abajo hacia arriba (*bottom-up*), de arriba hacia abajo (*top-down*) y combinaciones entre ellas. Una estrategia *bottom-up* funciona comenzando por el código, mientras que una estrategia *top-down* comienza por un objetivo de alto nivel. Cada estrategia se formula mediante la identificación de acciones para lograr un objetivo. Por ejemplo, en el caso de aplicar una estrategia hacia arriba, a partir del código vamos aplicando los mecanismos de agrupación y referencias cruzadas para producir estructuras de abstracción de nivel superior:

- La agrupación crea nuevas estructuras de abstracción de nivel superior a partir de estructuras de nivel inferior.
- Las referencias cruzadas enlazan elementos de diferentes niveles de abstracción.

Esto ayuda a construir un modelo mental del programa en estudio en diferentes niveles de abstracción.

Acrónimos

CBS Component Based Software system

COTS Commercial Off-the Shelf

SGBD Sistema de Gestión de Bases de Datos

Bibliografía

Belady, L. A. y M. M. Lehman. «A model of large program development». Editado por Springer. *IBM Systems Journal* 1, n.º 15 (1976): 225-252.

Tripathy, Priyadarshi y Kshirasagar Naik. *Software Evolution and Maintenance*. EE.UU.: Johh Wiley, 2014. <https://learning.oreilly.com/library/view/software-evolution-and/9781118960295/c02.xhtml>.

Tema 3. Arquitectura software

Desarrollo de Software

Curso 2022-2023

3º - Grado Ingeniería Informática

Dpto. Lenguajes y Sistemas Informáticos

ETSIIT

Universidad de Granada

5 de junio de 2023



Tema 3. Arquitectura software

Contenidos

3.1. Introducción	5
3.2. Principios de desarrollo en la arquitectura software	6
3.3. Propiedades no funcionales de la arquitectura software	12
3.4. Principios/propiedades mixtos	14
3.5. Descripción arquitectónica del software: el estándar IEEE P1471	15
3.5.1. Objetivos	15
3.5.2. Ingredientes básicos	16
3.5.3. Entidades conceptuales	16
3.5.4. Cómo declarar un punto de vista	19
3.5.5. Cómo realizar una Descripción Arquitectónica	19
3.6. Descripción arquitectónica del software: una propuesta concreta que amplía el estándar IEEE P1471	21
3.6.1. El proceso de definición de la arquitectura	22
3.6.2. Cómo identificar y comprometer a las partes interesadas en el sistema	24
3.6.3. Cómo identificar las inquietudes	28
3.6.4. Catálogo de puntos de vista	30
3.6.5. Plantilla para definir un punto de vista	32
3.6.6. Descripción detallada de un punto de vista: el punto de vista contextual	33
3.6.7. Descripción detallada de otro punto de vista: el punto de vista funcional	46
3.6.8. Adición de “perspectivas” al estándar P1471 basado en puntos de vista	63
3.6.9. Descripción detallada de una perspectiva: la perspectiva de seguridad	70
3.6.10. Descripción detallada de otra perspectiva: la perspectiva de evolución	86
3.7. Conclusiones	100
Acrónimos	101
Bibliografía	103

Arquitectura software

En este tema abordaremos la parte del desarrollo del software que se refiere a la arquitectura. En la sección 3.1 haremos un repaso del concepto de arquitectura del software y su evolución histórica. En la sección 3.2 estudiaremos los principios o técnicas de aplicación general en el desarrollo de software, y su relación con los patrones arquitectónicos. En la sección 3.3 estudiaremos las propiedades no funcionales de aplicación general en el desarrollo de software y, de nuevo, su relación con los patrones arquitectónicos. Destacaremos en la sección 3.4 un principio funcional y una propiedad no funcional fuertemente relacionados. La sección 3.5 la dedicaremos a presentar modelos actuales de especificación de arquitecturas. Para las dos primeras secciones, salvo otras referencias, nos basaremos en el libro *Software architecture*, de Mary Shaw and David Garlan.¹ En la tercera sección, después de exponer el estándar IEEEP1471, siguiendo sobre todo el trabajo de Richvid Hilliard *Using the UML for Architectural Description*,² utilizaremos la propuesta de Nick Rozansky³ en su libro *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition* como concreción del estándar de IEEE. También se usará alguna otra bibliografía adicional que se indicará en cada lugar, como el libro *Documenting Software Architectures: Views and Beyond, Second Edition* de Judith Stafford *et al.*⁴

3.1. Introducción

La *arquitectura software* es el nivel más elevado en el diseño software, que se refiere a la estructura general del sistema: los componentes que contiene y la relación entre ellos mediante conectores, así como los patrones arquitectónicos o estilos que guían estas relaciones y restricciones de los mismos.

La arquitectura incluye también las estructuras globales de control; los protocolos para la comunicación, sincronización y acceso a los datos; la forma de distribuir responsabilidades a los distintos elementos de diseño; la distribución física; el escalado y el rendimiento; la

¹Mary Shaw y David Garlan, *Software Architecture* (New Jersey: Prentice Hall, 1996).

²Rich Hilliard, «Using the UML for Architectural Description», ed. por Springer, *Proceedings of UML'99, Lecture Notes in Computer Science 1723* (1999).

³Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition* (Addison-Wesley Professional, 2011), <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

⁴Judith Stafford *et al.*, *Documenting Software Architectures: Views and Beyond, Second Edition* (EE.UU.: Addison-Wesley Professional, 2010).

evolución, etc.

Un *componente* es una unidad que proporciona alguna funcionalidad claramente distinguible de otras, o un almacén de datos. Algunos ejemplos de componentes son clientes, servidores, bases de datos, filtros, capas, tipos abstractos de datos, etc. Se pueden considerar como subsistemas si son simples o como sistemas completos si son compuestos (es decir, el componente tiene a su vez otros componentes que forman parte de él). El que los componentes puedan ser considerados subsistemas, o incluso sistemas, independientes, permite que puedan ser reutilizados, formando parte de otros sistemas.

Un *conector* es el elemento arquitectónico que modela y controla la interacción entre los componentes. Algunas interacciones son muy comunes y sencillas, como las llamadas a procedimientos y el acceso a variables compartidas, mientras que otras son más complejas, tales como los protocolos cliente-servidor, protocolos de acceso a las bases de datos, multidifusión asíncrona de eventos y flujo de datos por tubería (pipe data stream).

La arquitectura software muestra además cómo es la relación de la estructura con los requisitos del sistema al nivel más abstracto, el nivel arquitectónico, incluyendo requisitos no funcionales de calidad como capacidad, rendimiento, consistencia. Para guiar estas relaciones y las restricciones de los mismos se utilizan los *estilos* o *patrones arquitectónicos*.

3.2. Principios de desarrollo en la arquitectura software

Se trata de unos principios generales en el diseño de la arquitectura software de un sistema, independientes de los métodos concretos de desarrollo. En algunos casos puede haber conflicto en la aplicación de los principios y hay que priorizarlos. Aunque estos principios deben aplicarse desde la arquitectura software –nivel más alto en el diseño–, su aplicación debe continuar cuando se pasa a niveles más detallados de diseño. Así, los patrones arquitectónicos y de diseño aplican estos principios y también, cuando es necesario, hacen prevalecer unos principios sobre otros. Seguimos aquí de forma bastante aproximada la lista dada por la *banda de los cinco*.⁵

Abstracción La abstracción, en general, es una operación intelectual por la que los seres humanos extraemos un conjunto reducido de características de un ente para reducir la complejidad del mismo, de tal forma que sean suficientes para el desafío o problema que nos interesa. En arquitectura software la abstracción se realiza sobre los elementos del mundo real que se quieren modelar y sobre los distintos elementos software definidos para diseñar (componente, objeto, entidad, etc.). Un ejemplo de abstracción en OO es la herencia. Un ejemplo de patrón arquitectónico basado en la abstracción es el patrón *Capas*. Un ejemplo de patrón de diseño basado en la abstracción es el patrón *Factoría Abstracta*. Es muy importante ser capaz de extraer lo necesario para el problema en cuestión, ni más ni menos (ver Figura 3.1).

⁵Frank Buschmann et al., *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns* (Wiley Publishing, 1996), <https://learning.oreilly.com/library/view/pattern-oriented-software-architectur/e/9781118725269/>.

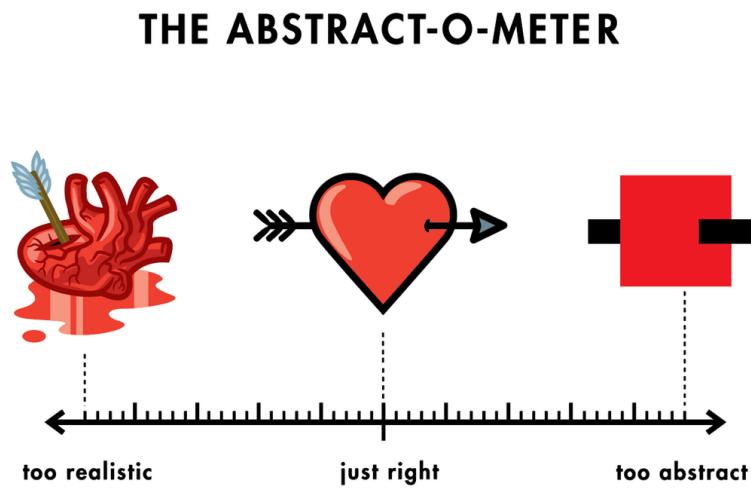


Figura 3.1: Ejemplo explicativo de tres formas de abstracción, siendo solo correcta la intermedia. [Fuente: <https://computersciencewiki.org/index.php/Abstraction>]

Modularización o *Divide y vencerás* Se trata de una forma de aplicar el principio anterior de reducción al justo nivel de complejidad mediante la división en partes del ente que quiere modelarse, pudiendo conllevar varios niveles de división o abstracción situándonos en el nivel que más convenga en cada momento según la tarea a realizar. En desarrollo software la modularización se puede aplicar solo a nivel lógico (por ejemplo los procedimientos y funciones en la programación estructurada, las división de responsabilidades en clases en cualquier lenguaje OO, los módulos Ruby o los espacios de nombres de C) o también a nivel físico (por ejemplo los paquetes Java). Los primeros programas, realizados en lenguaje máquina o ensamblador, eran a menudo programas monolitos, con una única secuencia de órdenes que podía ser muy larga. En seguida se empezaron a definir lenguajes de alto nivel, con ciertas características de modularización, como las funciones en lenguajes como Fortran o una parte para definición de datos en lenguajes como COBOL. La Figura 3.2 muestra la diferencia entre el diseño monolítico y el diseño modular. El diseño OO es siempre modular por modelar un sistema como un conjunto de clases o de objetos, salvo el caso de un programa con una sola clase o un solo objeto (como puede ser un “Hola mundo”). Por tanto, todos los patrones de diseño, por ser pensados para OO, utilizan la modularización. Asimismo, todos los patrones arquitectónicos parten de uso de la subdivisión de responsabilidades entre distintas partes (o “módulos”, en sentido genérico) en el sistema modelado (llamados componentes, objetos, paquetes, módulos, etc. según el enfoque y el lenguaje usado de programación).

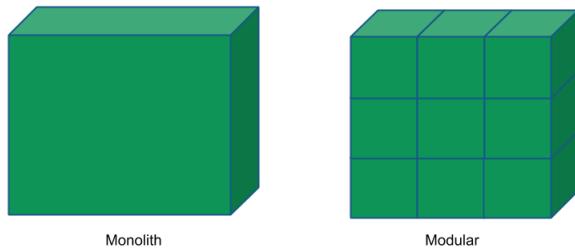


Figura 3.2: Desarrollo monolítico vs. modular [Fuente: <https://medium.com/@caitlinjeespn/modularization-in-software-engineering-1af52807ceed>].

Encapsulación La encapsulación asume la modularización, y consiste en separar o aislar un grupo de elementos del resto, estableciendo fronteras claras para delimitar los elementos separados. El que las fronteras estén claras no significa que los elementos externos no puedan ver el interior (ver Figura 3.3). Un ejemplo de encapsulación es la clase en OO, que agrupa atributos y métodos del ente que representa y los separa del resto del modelo, o los objetos que la instancian que tienen sus correlativos estado y comportamiento. Desde fuera es posible ver el interior, o parte de él (por ejemplo se puede consultar el estado del objeto). Un ejemplo de patrón arquitectónico que destaca la encapsulación es el patrón *Tubería y Filtros*. Un ejemplo de patrón de diseño que usa la encapsulación es el patrón *Observador* aunque en general, cualquier patrón de diseño usa la encapsulación, pues son patrones definidos especialmente para lenguajes OO, y en éstos, cada objeto –y cada clase en lenguajes con clases, que son la mayoría de los lenguajes OO⁶– forma su propia cápsula.

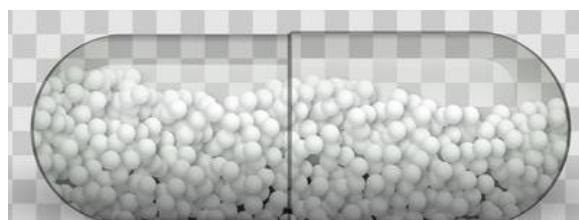


Figura 3.3: Medicamento en cápsula. Es un ejemplo que muestra que la delimitación de fronteras no tiene por qué impedir que desde fuera se pueda ver el interior. [Fuente: <https://es.dreamstime.com/sistema-de-la-p%C3%ADldora-c%C3%A1psula-aislado-en-fondo-transparente-ilustraci%C3%B3n-del-vector-image115117504>]

Ocultación de información La ocultación de la información asume la encapsulación, tratando de evitar que elementos vecinos de otro elemento puedan ver cierta información del

⁶Una excepción muy conocida de lenguaje OO que no usa clases en Java Script.

mismo, incluso si tienen permitida cierta interacción con el mismo (los clientes, en desarrollo software), de forma que la ocultación es solo parcial. Esto se puede hacer por razones de seguridad, confidencialidad o por facilitar la interacción a los clientes, y podrían ocultarse distintas partes según el cliente. Un ejemplo en OO es el uso de modificadores de acceso (privado para ocultar a cualquier cliente, protegido para ocultar según la relación del cliente con el objeto y el lenguaje de programación, etc.). Un ejemplo de patrón de diseño que usa este principio es el patrón *Fachada*. Como hemos visto también para la encapsulación y por las mismas razones, cualquier patrón de diseño va a usar este principio. Un ejemplo de patrón arquitectónico que usa este principio es el patrón *Fachada*.



Figura 3.4: Malla de ocultación. Ejemplo de una valla separadora (encapsulación) y una malla para ocultar (ocultación).



Figura 3.5: Ejemplo de encapsulación y ocultación parcial en una píldora, como símil con los objetos en OO [Fuente:<https://sites.google.com/a/innovavirtual.org/tecceilpvi/home/vi-ciclo-2014/poo/encapsulamiento>].

Bajo acoplamiento y alta cohesión Este principio asume también la modularización y establece que ésta debe hacerse manteniendo un bajo nivel de interacción entre elementos pertenecientes a distintos módulos (bajo acoplamiento) y fuerte nivel de interacción entre elementos pertenecientes al mismo módulo.

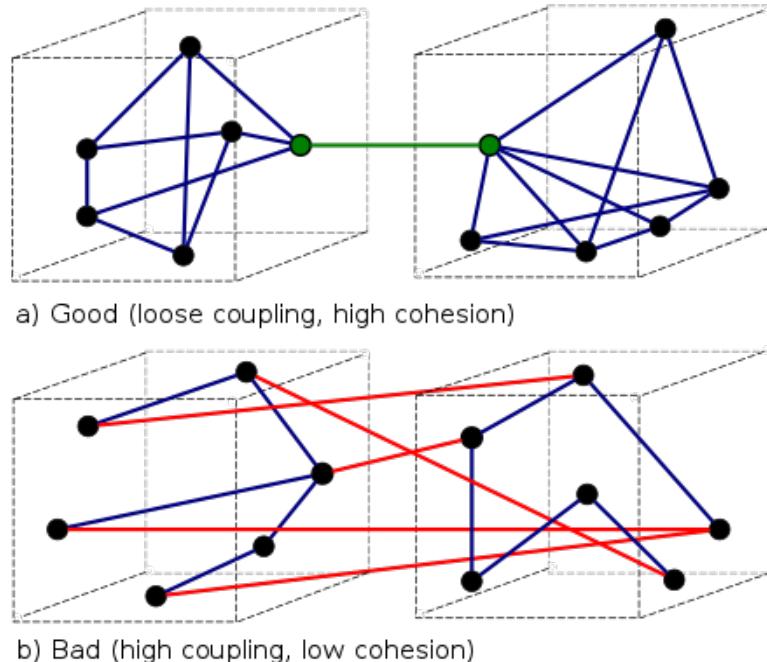


Figura 3.6: Ejemplo donde se muestra este principio, y la relación entre los conceptos de acoplamiento y cohesión [Fuente:[https://en.wikipedia.org/wiki/Coupling_\(computer_programming\)](https://en.wikipedia.org/wiki/Coupling_(computer_programming))].

Separación/agrupación (modularización) por responsabilidades e intereses Una forma de garantizar el principio anterior es guiando la modularización según intereses/responsabilidades, de forma que cada módulo (en sentido general) agrupe elementos con intereses y responsabilidades relacionados y se separen en distintos módulos intereses y responsabilidades no relacionadas. En desarrollo software, los intereses comunes o responsabilidades se refieren a tareas que deben trabajar unidas para proporcionar cierta funcionalidad a un sistema y deben así formar parte de una misma agrupación o módulo.



Figura 3.7: División de responsabilidades en un equipo humano. Cada persona se especializa en un área, asumiendo un conjunto de responsabilidades relacionadas entre sí para proporcionar un servicio de calidad en esa área [Fuente: https://es.123rf.com/photo_26977632_el-trabajo-en-equipo-de-los-profesionales-de-consultor%C3%ADa-de-negocio.html].

Suficiencia y necesidad El conjunto de elementos que forman parte de una abstracción es suficiente si captura todos las propiedades o elementos del ente real que son necesarios para representar el ente real de forma apropiada al propósito del modelo, de forma que sin añadir más elementos el modelo es eficaz. El conjunto de elementos que forman parte de una abstracción es necesario si todas las propiedades o elementos del ente real capturados por el modelo son necesarios para el mismo, de forma que si quitáramos alguno de ellos el modelo dejaría de ser eficaz. En la Figura 3.1 vista anteriormente, se puede ver que solo el modelo intermedio es suficiente y necesario. El modelo de la izquierda es suficiente pero no necesario (sobran elementos, no falta ninguno) y el modelo de la derecha es necesario pero no suficiente (faltan elementos, no sobra ninguno). El desarrollo de software, un modelo o una parte del mismo (componente, objeto, etc.) es suficiente y necesario si tiene solo y nada más que los elementos requeridos para cumplir con sus responsabilidades.

Separación de interfaz e implementación Se trata de poner en diferentes lugares el qué y el cómo de un elemento, de forma que la accesibilidad al cómo sea más restrictiva y se pueda cambiar el cómo sin afectar a los elementos que se relacionan con él. En desarrollo de software significa mantener en ficheros distintos la identificación de un elemento (la declaración de lo que hace) y su especificación (la descripción de cómo lo hace) de manera que por defecto se oculte la especificación al cliente de dicho elemento. Esto facilita el cambio de la especificación pues los clientes del elemento no quedan afectados. Un ejemplo en C y C++ es la separación de cabeceras y contenidos (ficheros .h y ficheros .c o .cpp) y otro ejemplo son las interfaces Java (ver Figura 3.8). Un ejemplo de patrón de diseño para seguir este principio en lenguajes que no lo ofrecen, es el patrón *Puente*.

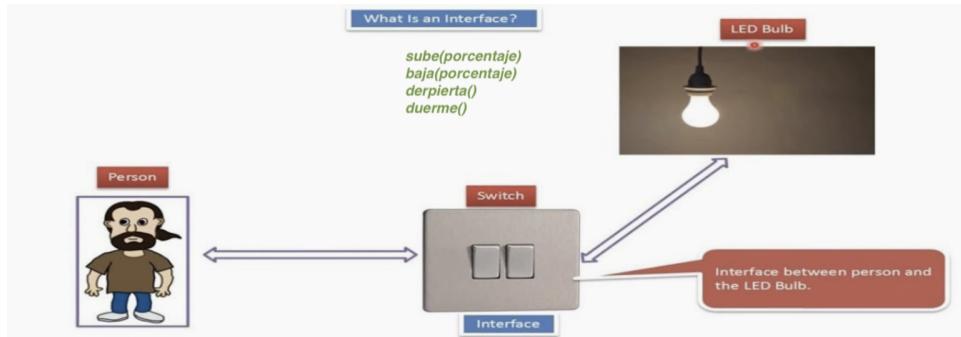


Figura 3.8: Ejemplo de interfaz con los métodos *sube*, *despierta*, *baja* y *duerme*. [Fuente: <https://knowitinfo.com/what-is-java-interface/> con modificaciones.]

La Figura 3.9 muestra un diagrama de clases (notación [Unified Modeling Language \(UML\)](#)) donde cada clase representa un principio de los vistosaquí, de manera que la relación “es un” que se da entre ellos (o –siendo principios–, la relación “asume” o “extiende”), se refleja como herencia.

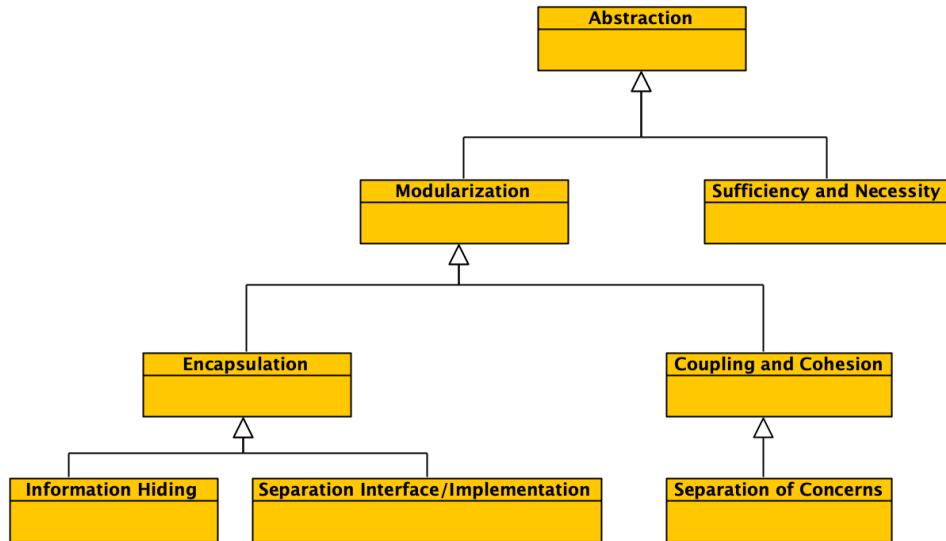


Figura 3.9: Relación entre los distintos principios de diseño arquitectónico.

3.3. Propiedades no funcionales de la arquitectura software

Las propiedades no funcionales impactan en la calidad del software tanto como las funcionales y, en especial en sistemas complejos y de larga vida, es fundamental tenerlas muy en cuenta. Afectan tanto en el desarrollo como en el mantenimiento, el funcionamiento general y el uso de los recursos computacionales.

Interoperabilidad Se trata de la facilidad con la que el software interacciona con otros elementos de su entorno y para ello se requiere un diseño bien definido de la funcionalidad y las estructuras de datos que los otros sistemas deben poder usar. Un ejemplo de patrón arquitectónico que considera bien esta propiedad es el patrón *Broker*.

Intercambiabilidad Se trata de la facilidad con la que el software desplegado pueda cambiar durante su vida, en especial, el que ha sido diseñado para largo tiempo. Se pueden distinguir cuatro aspectos de esta propiedad:

1. Mantenibilidad.- Se trata de la facilidad del software para que puedan ser corregidos errores detectados en él (el llamado *mantenimiento correctivo*). Dos ejemplos de patrones arquitectónicos que favorecen la mantenibilidad son los patrones *Reflexión* y *Puente*.
2. Extensibilidad.- Se trata de la facilidad con la que se pueda modificar para adaptarse a elementos externos (contexto) cambiantes y para añadir funcionalidad al software desplegado (los llamados *mantenimiento adaptativo* y *mantenimiento perfectivo* respectivamente).
3. Reestructuración.- Se trata de la facilidad con la que se pueden mover elementos entre distintos módulos o subsistemas, lo cual implica también cambios en los clientes de los mismos.
4. Portabilidad.- Se trata de la facilidad con la que el software puede adaptarse a distintas plataformas hardware, interfaces de usuario, sistemas operativos, lenguajes de programación o compiladores. Para lograrlo es necesario reunir aparte componentes especiales tales como librerías del sistema y de interfaces de usuario.

Eficiencia Se refiere a la capacidad del sistema para usar los recursos disponibles y el impacto que esto tiene sobre los tiempos de respuesta, rendimiento y consumo de memoria. Para ello no solo hay que usar algoritmos sofisticados, sino que es sobre todo necesario una correcta distribución de responsabilidades entre los componentes para garantizar un bajo acoplamiento y una alta cohesión. Algunos patrones, al priorizar principios funcionales u otras propiedades no funcionales puede producir bajas de la eficiencia. El patrón arquitectónico *Transportista-Receptor* prioriza esta propiedad.

Fiabilidad Se trata de la capacidad para que un sistema funcione a pesar de errores, tanto del sistema o de un uso incorrecto del mismo. Tiene dos modalidades:

1. Tolerancia a fallos.- Se trata de la capacidad de un comportamiento correcto en caso de errores y de la reparación interna, como desactivación de componentes afectados en software distribuido, hasta su reparación.
2. Robustez.- Se trata de la capacidad de proteger un sistema de usos incorrectos, y, en el caso de que sea necesario pararlo, hacerlo de forma definida.

Una forma de garantizar esta propiedad es añadiendo redundancia a propósito o componentes de monitorización y gestión de eventos. Un ejemplo de patrón arquitectónico que da prioridad a esta propiedad es el patrón *Maestro-Esclavo*.

Testabilidad Se trata de facilitar la capacidad de poder evaluar la correctitud de un sistema software, algo que suele ir unido a una mejor detección y corrección de errores y a la integración de componentes y código de depuración. Algunos patrones estructurales vistos facilitan la testabilidad, aunque no la traten directamente. Uno es por ejemplo el patrón *Procesador de Comandos* al facilitar testabilidad a nivel de interfaz de usuario (con ficheros logs y respuestas a las órdenes del usuario). Otro ejemplo es el patrón *Broker* pues separa componentes clientes y servidores, aunque por otro lado para mantener la independencia introduce componentes adicionales y la testabilidad se complica al tener que probarlos si se quieren depurar errores en el envío de un mensaje del cliente al servidor.

3.4. Principios/propiedades mixtos

Detallamos aquí un principio funcional (simplicidad) y una propiedad no funcional (reusabilidad) como formando una sola unidad, por estar fuertemente relacionados.

Simplicidad (Primitiveness), Reusabilidad y Parsimonia .- La simplicidad establece que la solución preferente a un problema, entre todas las correctas, sea la más sencilla. La reusabilidad se refiere a la capacidad de aplicar un solución a un problema similar. La parsimonia se define en contextos de soluciones correctas muy complejas y establece que la solución a un problema tenga en cuenta tanto la correctitud de la solución como la complejidad (ver Figura 3.10). En desarrollo de software, simplicidad significa optar siempre por las implementaciones más sencillas. Reusabilidad tiene dos perspectivas: desarrollo software *con reúso* (hacer software que haga uso de elementos que ya existen: proyectos anteriores, librerías comerciales, componentes, etc.) y desarrollo software *para reúso*. Este último significa optar por implementaciones que valgan para dar solución a problemas similares futuros. La parsimonia significaría, en soluciones a problemas software deterministas, mantener un equilibrio entre capacidad de reusabilidad futura (*para reúso*) o generalización y sencillez de la solución.

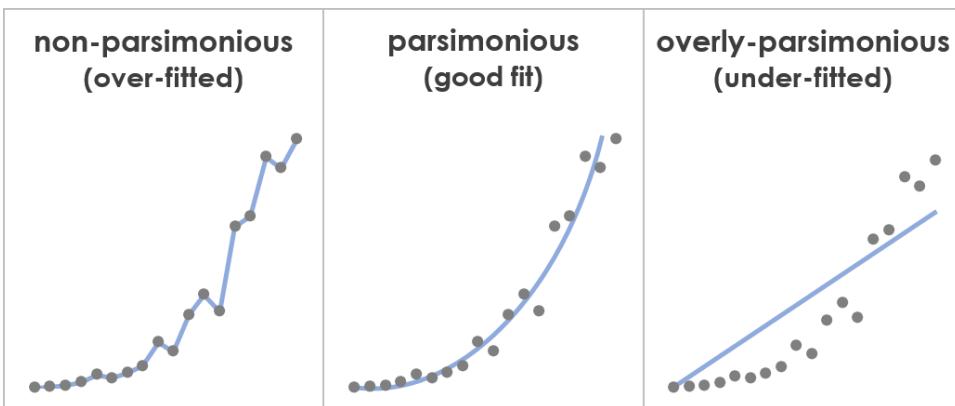


Figura 3.10: Ejemplo de modelo gráfico sobre-ajustado o no parsimonioso (izquierda), parsimonioso (centro) y sub-ajustado (derecha) [Fuente: <https://effectiviology.com/parsimony/>].

3.5. Descripción arquitectónica del software: el estándar IEEE P1471

En esta sección nos centraremos en la descripción del software a nivel arquitectónico, bajo el estándar IEEE P1471 (actualizado en 2011 por ISO/IEC/IEEE42010), que considera la aplicación de distintos puntos de vista según las partes interesadas en un sistema software.

NOTA: Para evitar numerosas citas literales, debe tenerse en cuenta que el texto de este apartado ha sido traducido, extraído o resumido del trabajo de Rich Hilliard *Using UML for Architectural Description*.⁷

En 1999 surgió un estándar (*prácticas recomendables*⁸), el P1471, elaborado por el IEEE Architecture Working Group^{9 10} para regular las arquitecturas de sistemas software *intensivos*¹¹. Este estándar regula en definitiva la manera en la que debe hacerse la *descripción arquitectónica (DA)* de un sistema software.

3.5.1. Objetivos

Los objetivos de IEEE para el P1471 son:

1. Asumir una interpretación amplia del concepto de arquitectura en sistemas software intensivos, es decir, sistemas basados en computadores, incluyendo aplicaciones software, sistemas de información, sistemas embebidos, sistemas de sistemas, líneas de

⁷Hilliard, «Using the UML for Architectural Description».

⁸En IEEE hay tres tipos distintos de estándar: estándar en sí, prácticas recomendables y guías.

⁹IEEE Architecture Working Group, *IEEE P1471/D5.0 Information Technology - Draft Recommended Practice for Architectural Description*, informe técnico (1999), <http://www.pithecanthropus.com/~awg/>.

¹⁰Este estándar fue modificado en 2011 por el estándar ISOIECIEE42010.

¹¹Un sistema software intensivo es un sistema software o un sistema general donde el software influye de forma fundamental en el diseño, construcción, desarrollo y evolución del sistema.

productos y familias de productos donde el software juegue un papel fundamental en el desarrollo, operación o evolución del sistema.

2. Establecer un marco de trabajo conceptual y un vocabulario para hablar de los aspectos arquitectónicos del sistema, de forma que todos puedan entenderlo. Esto supone ponerse de acuerdo en los significados de algunos términos como *arquitectura*, *Descripción Arquitectónica (DA)* y *vista*.
3. Identificar y declarar prácticas arquitectónicas sólidas, pues habiendo numerosas propuestas falta proporcionar las bases para que las propuestas puedan definirse, contrastarse y aplicarse.
4. Permitir la evolución de estas prácticas conforme evolucionan tecnologías que sean relevantes.- IEEE reconoce la rápida evolución de las prácticas software arquitectónicas, tanto a nivel industrial como de investigación, y por eso P1471 pretende ser un marco de referencia para que estas prácticas puedan comunicarse, documentarse y compartirse. Por ello, el marco debe ser suficientemente general para acoger las técnicas actuales y suficientemente flexible para que pueda evolucionar.

3.5.2. Ingredientes básicos

El P171 contiene tres ingredientes básicos:

1. Un conjunto de normas de definición de términos tales como *descripción arquitectónica*, *vista arquitectónica* o *punto de vista arquitectónico*.
2. Un marco conceptual que sitúa esos términos en el contexto de los múltiples usos posibles de DAs para la construcción, análisis y evolución de sistemas.
3. Un conjunto de requerimientos sobre una DA de un sistema.

P1471 no exige para una DA consideraciones sobre los sistemas, proyectos, empresas, procesos, métodos o herramientas, sólo que la DA cumpla con los estándares, que se definen con requerimientos (“debe cumplir con”). Así, es importante destacar que el estándar IEEE P1471 se ha desarrollado de forma que es independiente de la notación.

3.5.3. Entidades conceptuales

En P1471 hay un conjunto básico de entidades conceptuales que son usadas en la definición del estándar y que es necesario conocer:

- *Descripción arquitectónica*.- Colección de productos para documentar la arquitectura de un sistema. No especifica el formato o el medio pero sí unos requerimientos mínimos de contenidos que reflejen las prácticas actuales y el consenso industrial.

- *Parte interesada (stakeholder).*- Cualquier individuo, clase o componente externo, institución o rol interesado en el sistema. Algunos ejemplos son: cliente del sistema, usuarios, operadores, personal encargado del mantenimiento del sistema, desarrolladores, comerciales, etc.
- *Inquietud (concern).*- Cualquier tipo de preocupación sobre la arquitectura que tengan las partes interesadas en el sistema, como las garantías de seguridad, fiabilidad, o mantenibilidad.
- *Vista.*- Cada una de las partes en las que se divide una DA. P1471 no exige unas vistas concretas pues éstas dependen de la técnica usada, sino que lo deja a criterio de los usuarios del estándar. Las vistas son los mecanismos para separar los distintos intereses, tanto para reducir la complejidad del sistema como para ajustarse a las necesidades de las distintas partes interesadas. A su vez, cada vista puede estar compuesta de varios *modelos arquitectónicos*, cada uno con un esquema distinto de representación, por ejemplo un modelo puede usar un diagrama de clases Unified Modeling Language (UML) y otro un diagrama de componentes UML.
- *Punto de vista.*- Contiene las reglas por las que se rigen las vistas concretas. Un aspecto interesante es la idea de reusabilidad de los puntos de vista (que pueden recogerse en una *librería de puntos de vista*), de forma que puedan aplicarse a otros sistemas, considerándolos como referencias o patrones¹². Para elegir los puntos de vista debemos partir de un inquietud concreta y, a partir de esa inquietud, elegir un punto de vista con vistas concretas que se ajusten a ese punto de vista. P1471 no propone puntos de vista específicos, sólo que los que se elijan se entiendan y documenten bien. En la DA debe siempre quedar claro el punto de vista que se aplica en cada una de las vistas para hacer éstas inteligibles.

La Figura 3.11 representa el modelo conceptual del estándar P1471.

¹²De aquí deriva el concepto de patrón o estilo arquitectónico.

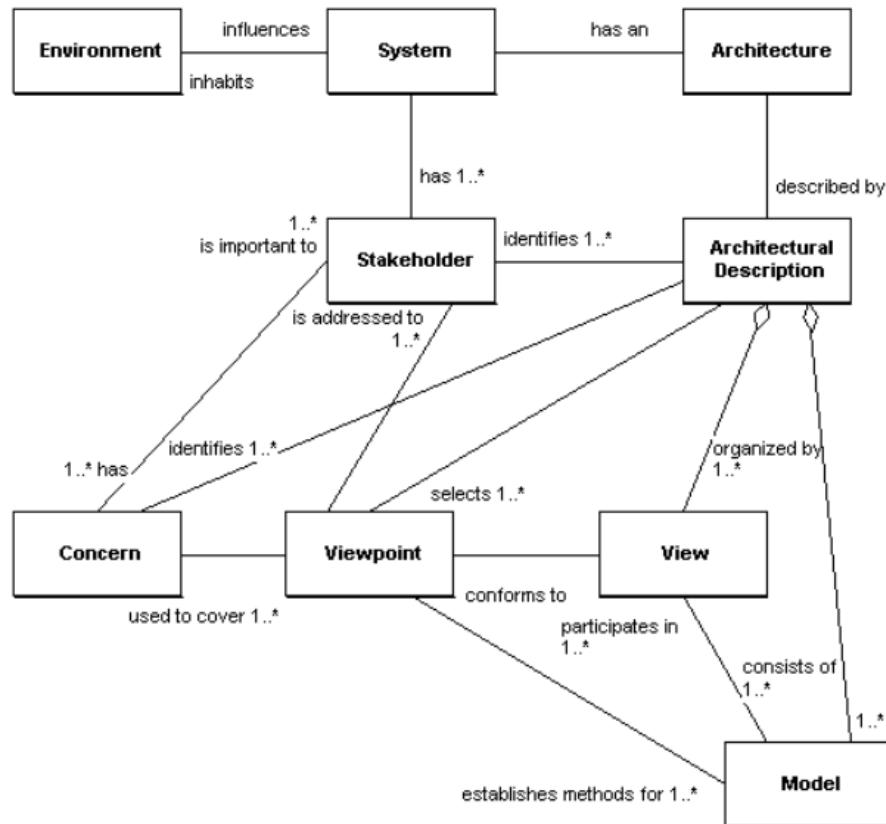


Figura 3.11: Modelo conceptual del estándar IEEE P1471. [Fuente:¹³]

CRITERIO DE CALIDAD 3.5.1: Seguridad

La seguridad software es la protección contra toda tipo de amenaza que pueda impedir el correcto funcionamiento del software, tanto debida a ataques intencionados como a accidentes o catástrofes.

CRITERIO DE CALIDAD 3.5.2: Fiabilidad

La fiabilidad software es la probabilidad de que el sistema funcione sin fallos debidos al diseño durante un período específico de tiempo. Garantizar la misma en software muy complejo se hace más difícil.

CRITERIO DE CALIDAD 3.5.3: Mantenibilidad

La mantenibilidad software se define como el grado en el que una aplicación puede comprenderse, repararse y mejorarse. Una baja mantenibilidad eleva de forma considerable el costo de un proyecto software.

Una DA debe identificar a sus partes interesadas y las inquietudes que tengan, y debe dar respuesta a todos estas inquietudes para que sea **completo**.

CRITERIO DE CALIDAD 3.5.4: Completitud

La completitud de una DA de un sistema software es la condición de que da respuesta a las inquietudes de todos las partes interesadas en el sistema.

3.5.4. Cómo declarar un punto de vista

Debe especificar los siguientes elementos:

- Nombre del punto de vista
- Partes interesadas con inquietudes abordados por este punto de vista
- Intereses/inquietudes abordados por este punto de vista
- Lenguaje, técnicas de modelado y métodos analíticos usados
- Fuente, si hay alguna, del punto de vista (autor, citas)

Otros aspectos no obligatorios:

- Cualquier tipo de método de comprobación de la consistencia o completitud incluido por el método usado que pueda aplicarse en los modelos concretos elaborados para una vista
- Cualquier técnica de evaluación o análisis que pueda aplicarse a los modelos elaborados para esa vista
- Cualquier heurística, patrón (en el sentido de estilo o patrón arquitectónico) u otra guía que sirva de ayuda para sintetizar las vistas asociadas con el punto de vista o sus modelos

3.5.5. Cómo realizar una Descripción Arquitectónica

La forma de proceder es que el arquitecto software elija los puntos de vista arquitectónicos desde una lista predefinida de ellos (la librería de puntos de vista) en atención a cada uno de las inquietudes que el sistema debe cubrir. De forma alternativa puede escoger un patrón arquitectónico que de respuesta a cada inquietud concreta.

Una vez elegido cada punto de vista será necesario describirlo como se ha especificado anteriormente, incluyendo entre otros el lenguaje y métodos de modelado a usar.

Una herramienta de modelado (y lenguaje) muy apropiado es [UML](#), pues también está desarrollado de forma independiente del proceso, pudiéndose utilizarse para cualquier tipo de software que se quiera desarrollar. La guía del usuario de [UML](#) precisamente proporciona cinco distintos puntos de vista (aunque los llama vistas entrelazadas): dirigida por los casos de uso, centrada en la arquitectura, iterativa, de proceso incremental, . . .¹⁴

¹⁴Hilliard, «Using the UML for Architectural Description».

Hilliard¹⁵ propone cuatro formas distintas de usar **UML** para aplicar el estándar IEEE P1471:

1. *Listo para usar* (out of the box) o *predefinido*.- Se trata de usar **UML** como conjunto de herramientas de notación, de forma que para cada vista podemos usar uno o más diagramas **UML**. Se puede considerar cada tipo de diagrama como un posible lenguaje aplicable a un punto de vista, según la Tabla 3.1.
2. *Extensión ligera*.- Se trata de usar los mecanismos de extensión ligera que proporciona **UML** (etiquetas, estereotipos, restricciones) para añadir información en la **DA**. Se proponen dos posibilidades:
 - Usar extensiones de **UML**.- Se amplía **UML** con un conjunto concreto de estereotipos, valores etiquetados (restricciones), y restricciones concretas en un dominio de aplicaciones
 - Usar variantes **UML**.- El metamodelo **UML** se deja como está pero sobre él se construyen variantes.
3. *UML como marco integrador*.- Otra posibilidad es que el arquitecto software, o el equipo e empresa, adopten el metamodelo **UML** como marco integrador para hacer la **DA**, es decir, la **DA** se describa por completo con **UML** y los distintos puntos de vista y las vistas se relacionan entre sí a través de los distintos diagramas **UML**. Para ello hay que entender la relación entre el estándar P1471 y **UML**: en **UML** se considera el concepto de *técnica diagramática* (TD) que permite regular las distintas vistas. Así, una TD permite a los arquitectos software documentar un tipo concreto de diagrama y añadir extensiones. Con este enfoque, se definen los puntos de vista y se asocia a cada una de sus vistas uno o varios diagramas. Pero además se puede proporcionar notación para representar toda la **DA** de manera global, por medio de un diagrama que incluye a las partes interesadas, los distintos intereses, los puntos de vista y las vistas (véase como ejemplo la Figura 3.12). Asimismo se deben integrar las vistas de manera que se garantice la consistencia entre ellas. **UML** proporciona como mecanismo para lograrlo, la relación *refine* (*ref*) por la cual se representan los mismos elementos en distintos niveles de abstracción. Faltaría sin embargo una forma de identificar elementos que representan lo mismo en distintas vistas o puntos de vista. Por ejemplo, un elemento en una vista estructural puede ser un componente y en una vista de datos representarse como interfaz de acceso a los datos.
4. *Fuera de la ontología UML*.- A menudo nos encontraremos con puntos de vista para los que no existen diagramas **UML** y será más fácil usar otras notaciones o técnicas que extender **UML**. Algunos ejemplos son los diagramas de Gantt, de presupuestos y

¹⁵Hilliard, «Using the UML for Architectural Description».

organizativos para un punto de vista de gestión, o los modelos de bloques y de fallos si se trata de un punto de vista de la fiabilidad.

Punto de vista	Tipo de diagrama UML
Estructural	Diagrama de componentes; diagrama de clases
Conductual	Diagrama de interacción; diagrama de actividad; diagrama de estados
Usuario	Diagrama de casos de uso; diagrama de interacción
Distribución	Diagrama de despliegue; diagrama de interacción

Tabla 3.1: Relación entre los puntos de vista arquitectónicos y los tipos de diagrama UML [Fuente:¹⁶].

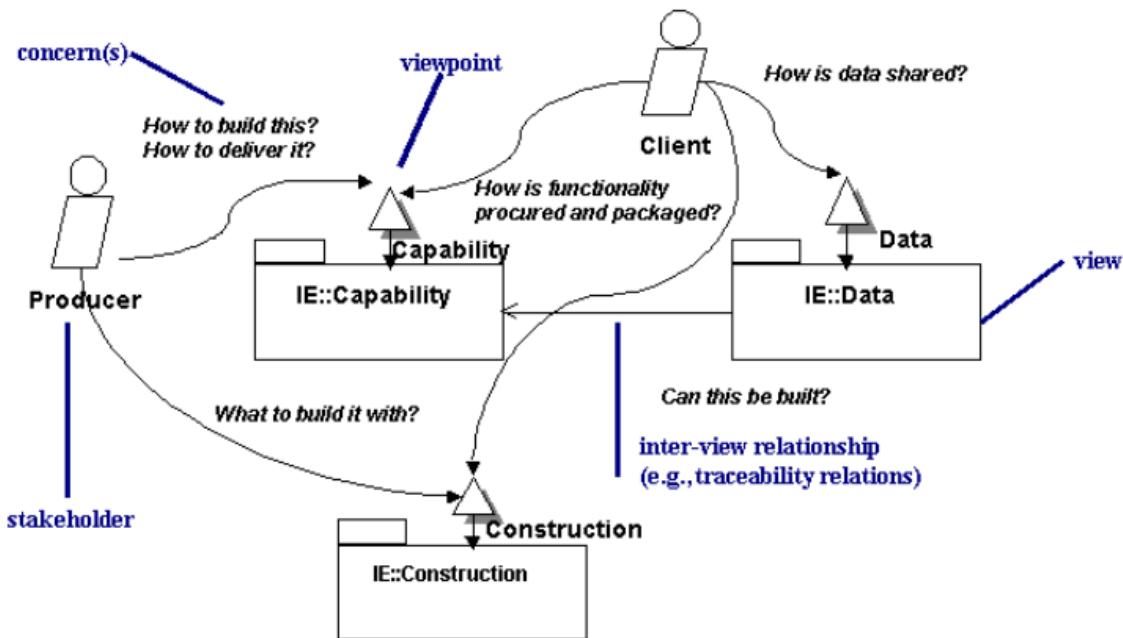


Figura 3.12: Un fragmento de una visión general de un sistema. Los iconos son las partes interesadas, las inquietudes se representan con flechas, los puntos de vista con triángulos y las vistas con paquetes. [Fuente:¹⁷]

3.6. Descripción arquitectónica del software: una propuesta concreta que amplía el estándar IEEE P1471

NOTA: Para evitar numerosas citas literales, debe tenerse en cuenta que para el resto de esta sección hemos traducido, extraído o resumido del trabajo de

¹⁶Rich Hilliard, «Using the UML for Architectural Description», ed. por Springer, *Proceedings of UML'99, Lecture Notes in Computer Science 1723* (1999).

Rozansky.¹⁸

La propuesta de este estándar se ha desarrollado por algunos autores con cierto detalle. Como ejemplo, veremos aquí la de Rozansky.¹⁹

3.6.1. El proceso de definición de la arquitectura

La definición de la arquitectura empieza muy pronto en el ciclo de vida del proyecto, cuando a menudo no están claros ni el ámbito ni los requisitos, lo que hace que sea un proceso bastante más flexible que los que se llevan a cabo en las fases siguientes.

Principios guía del proceso Rozansky considera que el proceso de realización de la DA debe considerar los siguientes aspectos, independientemente de la metodología de desarrollo que se use (ciclo de vida clásico –modelo de cascada–, enfoques iterativos –como prototipos–, metodologías ágiles, etc.):

- Debe ser dirigido por las inquietudes de las partes interesadas.
- Debe fomentar la comunicación entre el equipo técnico (decisiones arquitectónicas y principios) y las partes interesadas no técnicas.
- Debe realizarse de forma bien organizada (definiendo objetivos, y encadenando la salida de cada fase con la entrada de la siguiente).
- Debe ser pragmático y confrontarse con la realidad (falta de recursos, cambios de requisitos o del contexto o de la organización donde se utilizará).
- Debe ser flexible para adaptarse a las circunstancias particulares.
- Debe ser independiente de la tecnología, sin imponer de forma previa patrones arquitectónicos, estilos de desarrollo, de modelar, documentar, etc.
- Debe integrarse dentro del ciclo de desarrollo software que se elija.
- Debe usar buenas prácticas de ingeniería software, y estándares de gestión de calidad (como ISO 9001) para poder integrarse en enfoques existentes.

Resultados del proceso Además de obtenerse la descripción de una arquitectura sólida que permita una buena gestión de las fases de diseño y mantenimiento, también debe producir otros resultados:

- Ayudar a clarificar los requisitos y otras entradas, que a menudo las distintas partes interesadas no tienen bien establecidos antes de iniciarse este proceso de DA.

¹⁸Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*.

¹⁹Rozanski.

- Gestionar las expectativas de las partes interesadas, que a menudo están en conflicto, para llegar a soluciones que todos entiendan y acepten antes de avanzar más en el desarrollo del software.
- Identificar y evaluar las distintas opciones arquitectónicas, los pros y contras de cada una y justificar la solución adoptada.
- Describir de forma indirecta los criterios de aceptación de la arquitectura, como el cumplimiento de lo especificado en la DA.
- Crear un conjunto de entradas de diseño, que permitirá guiar e imponer restricciones en el proceso de diseño ayudando a garantizar la integridad de la arquitectura.

El contexto del proceso El proceso de definición de la arquitectura es el primer paso en la etapa de desarrollo software, tan intrincada con la anterior etapa (especificación de requisitos) y el siguiente paso en el desarrollo (diseño) que a menudo se representa el producto (la arquitectura) como el punto medio entre ambas fases, teniendo los productos de las otras fases (requisitos y software construido) fronteras poco delimitadas con la arquitectura (Figura 3.13). En la figura también se observa cómo el nivel de elaboración (anchura de cada producto) y el solapamiento de los productos aumentan con el nivel de detalle.

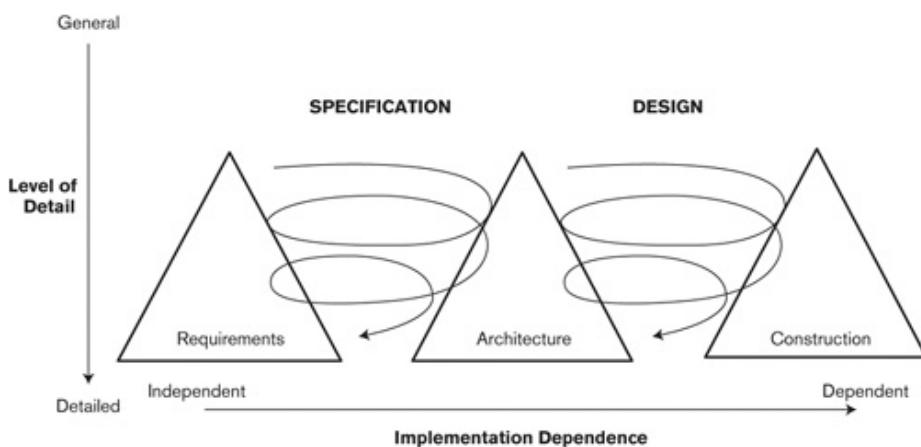


Figura 3.13: El contexto de definición de la arquitectura [Fuente:²⁰]

Actividades de soporte Se han identificado los siguientes actividades de soporte en el proceso de definición de la arquitectura:

- Identificar e involucrar a las partes interesadas (Sección 3.6.2).
- Llegar a un acuerdo sobre el ámbito, el contexto, las restricciones y los principios arquitectónicos base en los que apoyarse²¹.

²¹Para detalles sobre principios y toma de decisiones, consúltese Rozanski, Cap. 8.

- Identificar las inquietudes de las distintas partes interesadas ([3.6.3](#)).
- Dentro de la definición de la arquitectura:
 - Usar un catálogo de puntos de vista (con una plantilla para cada vista) y un catálogo de perspectivas para elegir las vistas y perspectivas transversales de calidad, respectivamente (Sección [3.6.4](#)).
 - Usar escenarios arquitectónicos (Sección [3.6.6](#),²²).
 - Usar estilos y patrones arquitectónicos, especialmente en la vista funcional pero también en otras (Tema 1,²³), para producir los modelos arquitectónicos, que formarán parte de la documentación de la arquitectura.
- Validar la arquitectura.

3.6.2. Cómo identificar y comprometer a las partes interesadas en el sistema

Rozansky ha detallado una guía sobre cómo identificar y comprometer a las partes interesadas en el sistema²⁴.

DEFINICIÓN 3.6.1: Parte interesada

Una parte interesada en la arquitectura de un sistema es un individuo, equipo, organización y tipo de ellos con algún interés en la realización del sistema.

Es muy importante identificar correctamente a los más importantes y a partir de ahí intentar resolver sus inquietudes o preocupaciones. Los grupos más importantes de partes interesadas son:

- Los más afectados por las decisiones arquitectónicas, como usuarios, operadores o los clientes que pagan por la realización del sistema.
- Los que influyen en la forma y en el éxito del desarrollo, como los clientes.
- Los que deben incluirse por razones organizativas o políticas, tales como los administradores, un equipo encargado de la gestión de la arquitectura global, o alguna persona con autoridad en la empresa.

No tener en cuenta las inquietudes de las partes interesadas más importantes desde el principio hará que cuando las planteen fuercen el rediseño, con un aumento significativo de los costes de desarrollo.

Una parte interesada que haga bien su trabajo debe:

²²Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*, Cap. 10.

²³Rozanski, Cap. 11.

²⁴Rozanski, Cap. 9.

- Estar informada y con la experiencia suficiente para tomar decisiones adecuadas.
- Comprometerse en participar en el proceso de desarrollo y ser capaz de tomar decisiones difíciles si llega el caso.
- Tener la autoridad necesaria para tomar decisiones y evitar que sean revertidas por otros en el futuro.
- Ser capaces de representar a todo un grupo con los mismos intereses y llegar a tener criterios compartidos.

Según el papel y las inquietudes, las partes interesadas se clasifican en distintos tipos. Dependiendo del sistema unos tendrán más importancia que otros, pero rechazar algún tipo dará problemas futuros. Los tipos propuestos son los siguientes:

- Clientes o adquisidores: supervisan que el sistema sea desarrollado
- Asesores: supervisan que el sistema se ajuste a los estándares y regulación legal
- Comunicadores: explican el sistema a otras partes interesadas mediante documentación y material de entrenamiento
- Desarrolladores: construyen y despliegan el sistema a partir de sus especificaciones, o lideran equipos que lo hagan
- Mantenedores: gestionan la evolución del sistema una vez en explotación
- Ingenieros de producción: diseñan, despliegan y gestionan los entornos hardware y software en los que el sistema se construirá, probará y ejecutará
- Proveedores: construyen y/o proporcionan el hardware, software o la infraestructura donde se ejecutará el sistema
- Equipo de apoyo: proporcionan apoyo a usuarios del producto o sistema cuando se ejecuta
- Administradores del sistema: ejecutan el sistema una vez lanzado
- Equipo de prueba: prueba el sistema para asegurar que está preparado para ser usado
- Usuarios: definen la funcionalidad del sistema y hacen uso del mismo

Es necesario mantener un equilibrio entre el número de partes interesadas y la facilidad para llegar a un consenso. El arquitecto debe gestionar el proceso de toma de decisiones y tener una idea clara de la importancia relativa de cada uno de los grupos de partes interesadas. Esto será necesario a la hora de defender una decisión arquitectónica ante partes interesadas que consideren que sus inquietudes están siendo ignoradas.

El propio arquitecto debe considerarse a sí mismo una parte interesada en el sistema.

Se proponen algunos ejemplos para aprender a identificar y elegir de forma apropiada a las partes interesadas.

EJEMPLO 3.6.1. Proyecto de desarrollo a partir de software comercial (*off-the-shelf*)

Estos proyectos requieren la selección, adaptación e implementación a partir de un paquete software, de forma que se desarrolla menos software que en un sistema tradicional. Pero el papel de las partes interesadas sigue siendo vital.

Como ejemplo, una empresa A, fabricante de hardware, quiere usar un sistema de planificación de recursos empresariales (Enterprise Resource Planning, ERP) para manejar mejor todos los aspectos de la cadena de producción, desde la petición hasta la entrega. Los gestores han considerado la construcción del sistema mediante un paquete software comercial *personalizable* (*custom off-the-shelf*, COTS) combinado con algún software de desarrollo propio para algunos aspectos más especializados de la funcionalidad. El nuevo sistema debe ser desplegado en un año, a tiempo para una reunión con los accionistas que están considerando una futura financiación de la empresa.

Los adquirientes del sistema incluyen al director gerente, que autorizará la financiación del proyecto, junto con el departamento de compras y los representantes informáticos, que evaluarán distintos paquetes ERP.

Los usuarios del sistema cubren un amplio rango de empleados internos, entre los que se incluyen los que procesan los pedidos, las compras, la financiación, la fabricación y la distribución.

Los desarrolladores, administradores del sistema, ingenieros de producción y mantenedores y personal del departamento de informática así como los asesores se elegirán desde el equipo de prueba de aceptación interna. Los comunicadores incluyen a los entrenadores internos, y el soporte es proporcionado por el equipo interno de ayuda (help desk), posiblemente en unión con los proveedores COTS.

EJEMPLO 3.6.2. Proyecto de desarrollo de un producto software

Un producto software es usualmente desarrollado por un proveedor especializado, con el desarrollo a menudo parcialmente financiado por inversores externos. Los usuarios previstos del producto estarán en otras organizaciones que, se espera, comprarán el producto una vez que esté completo. Las partes interesadas para tales proyectos a menudo se reparten entre varias organizaciones.

Como ejemplo, una empresa B, un proveedor de software educativo, quiere desarrollar un producto que será utilizado por los profesores universitarios para administrar sus horarios de clases. La empresa B ha formado una sociedad con una universidad local y ha obtenido algunos fondos de capital de riesgo para el producto. El sistema se ejecutará en PCs y será económico y fácil de usar.

Los adquirientes en este caso incluyen directores gerentes y jefes de producto de la empresa B, el socio educativo (la universidad), y representantes de los capitalistas de riesgo.

Los usuarios del sistema son profesores universitarios y personal administrativo; tenga en cuenta que estos usuarios en realidad no existen todavía como tales porque nadie ha comprado el producto todavía. La representación del usuario deberá obtenerse de alguna otra manera (por ejemplo, hablando con algunos usuarios potenciales del producto).

Los desarrolladores y mantenedores son personal de desarrollo de productos de la empresa B, y los asesores son tomados de las tres empresas asociadas. No hay partes interesadas que sean administradores del sistema real en este ejemplo (esto debe tenerse en cuenta en la arquitectura, por ejemplo, haciendo que el sistema se autogestione). La empresa B y/o las universidades que compran el producto pueden proporcionar el personal de mantenimiento.

Los comunicadores incluyen autores técnicos de la empresa A que escriben la guía de usuario.

Los ingenieros de producción proporcionan los entornos de desarrollo y prueba para la empresa B. También proporcionan y administran la infraestructura para fabricar CDs de productos y distribuir actualizaciones de software a los usuarios a través de Internet.

EJEMPLO 3.6.3. Desarrollo subcontratado

Un proyecto de desarrollo subcontratado involucra a una organización que utiliza los servicios de otra para proporcionar sistemas o servicios que normalmente proporcionaría con sus propios recursos internos. Estos proyectos dan como resultado que las partes interesadas que normalmente se encontrarían dentro de la organización adquiriente se encuentren dentro de la organización de servicios externos. Esto puede dificultar la identificación e interacción con estas partes interesadas.

La empresa C, una empresa financiera consolidada, desea expandir su presencia en Internet con la capacidad de comercializar una gama de servicios financieros a clientes últimos. Estos servicios están dirigidos a los residentes del país donde tiene su sede la empresa C, así como a algunos clientes internacionales. La empresa C planea contratar el desarrollo y uso del sistema a un desarrollador web establecido.

Los adquirientes incluyen directores gerentes que autorizarán la financiación del proyecto. Los usuarios incluyen clientes ordinarios, que accederán al sitio web público (como en el ejemplo anterior, estos todavía no existen), junto con el personal administrativo interno, que llevará a cabo sus funciones de trabajo de respaldo.

Los administradores del sistema son personal de la empresa de desarrollo web. Los asesores incluyen el personal interno de contabilidad y abogados de la empresa C, así como los reguladores financieros externos de cualquier país en el que la empresa C desee comerciar.

Los comunicadores, los ingenieros de producción y el personal de mantenimiento son proporcionados por la empresa C y/o la empresa de desarrollo web.

Cuando las partes interesadas aún no existan como grupo, se pueden identificar *partes interesadas de proximidad (proxy stakeholders)*, personas o grupos que representen las inquietudes de las partes interesadas reales y aseguren que se les da la misma importancia que a las inquietudes de las otras partes interesadas. Por ejemplo, los clientes potenciales pueden ser representados por los gestores de productos del grupo de marketing, que tengan los resultados de pruebas de marketing, o por miembros de la población de usuarios objetivo que estén dispuestos a involucrarse en la concepción de los productos.

3.6.3. Cómo identificar las inquietudes

El concepto de inquietud (del inglés, concern) está definido por Rozansky²⁵ de una forma más detallada que la que da el estándar IEEE P1471.

DEFINICIÓN 3.6.2: inquietud (concern)

Una inquietud sobre una arquitectura es un requisito, objetivo, restricción, intención o aspiración que una parte interesada tenga para dicha arquitectura.

²⁵Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*, Cap. 6.

Por tanto, no sólo incluyen a los requisitos funcionales y no funcionales, que son inquietudes específicas, sin ambigüedad y medibles, sino que también incluyen otras inquietudes que se formulan de forma mucho más vaga y poco definida pero que no por ello son menos importantes para el interesado que las formula. Incluso pueden ser más importantes que algunos requisitos específicos.

EJEMPLO 3.6.4. Mantener la reputación del servicio proporcionado

Un minorista tiene una gran reputación en la calidad del servicio y la respuesta que da a sus clientes. Esto hace que tenga algunos objetivos y aspiraciones para una nueva tienda en línea que quiere construir:

- Los valores, la ética y la reputación del minorista deben reflejarse en la apariencia y el funcionamiento de la tienda en línea y sus procesos de respaldo.
- En todo momento, el sitio web debe tratar de presentar una cara “humana” al cliente (incluso aquellas partes de la misma que están completamente automatizadas).
- La tienda en línea debe ser fácil de usar para los clientes que tienen una experiencia limitada con los ordenadores y con el comercio electrónico.
- La tienda en línea debe ser *responsiva* (rápida de cargar y responder a las acciones del cliente) independientemente de la velocidad de conexión a Internet del cliente.
- La tienda en línea debe cubrir todos los aspectos de la experiencia de compra, incluido un catálogo actualizado y navegable, un sistema seguro de compra en línea, el seguimiento de un pedido y la gestión de devoluciones.

Excepto el último elemento, ninguno de los otros pueden considerarse requisitos formales y medibles, y el último es realmente una declaración del ámbito que se desea cubrir. Sin embargo, si el sistema no cumple con estos objetivos y aspiraciones, probablemente será visto como un fracaso.

Rozansky define además otros dos criterios de clasificación de las inquietudes. El primero las clasifica según si:

- se refieren al problema (inquietudes enfocadas en el problema), como la filosofía o estrategia del negocio, y responden a las preguntas por qué y qué, o si
- se refieren a la solución (inquietudes enfocadas en la solución), casi siempre derivadas, de forma directa o indirecta de las primeras, como la estrategia TIC, que deriva de la filosofía del negocio, y que responden a las preguntas cómo y con qué.

El segunda las clasifica según si:

- son agentes que influyen y dirigen la toma de decisiones en una cierta dirección (por ejemplo, un objetivo para el negocio o para la tecnología), o si

- son agentes restrictivos sobre las decisiones que puedan tomarse (por ejemplo, un estándar o una política de actuación que estemos obligados a seguir).

Se pueden ver ejemplos de inquietudes clasificadas según estos dos criterios en la Figura 3.14.

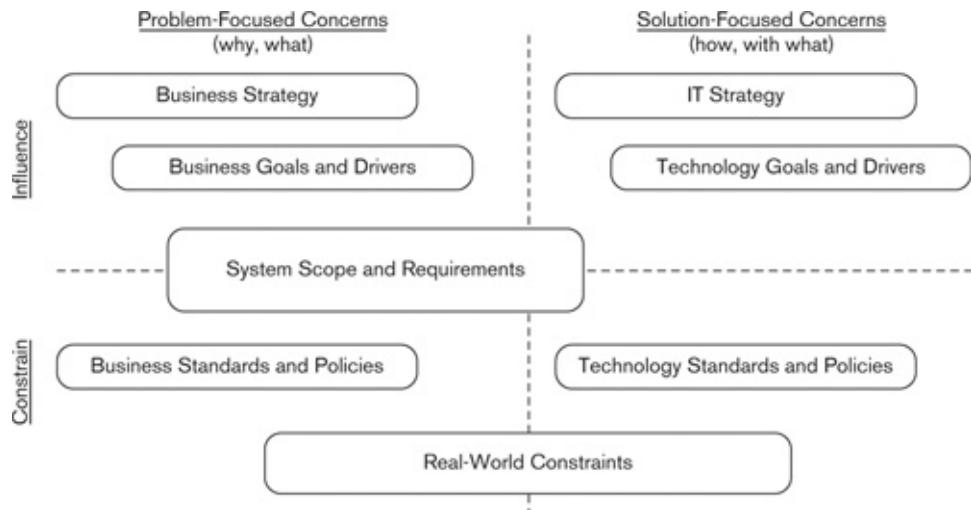


Figura 3.14: Inquietudes según dos criterios de clasificación. [Fuente:²⁶]

3.6.4. Catálogo de puntos de vista

Asimismo, Rozansky ha propuesto un catálogo con siete distintos puntos de vista a tener en cuenta en todo sistema software²⁷ (ver Figura 3.15):

- Punto de vista contextual.- Describe las relaciones, dependencias e interacciones entre el sistema y su entorno (personas, sistemas y otras entidades externas con las que interactúa). Este punto de vista es el que requieren muchas partes interesadas en el sistema, ayudándoles a entender las responsabilidades del sistema y su relación con la institución.
- Punto de vista funcional.- Describe los elementos funcionales del sistema, sus responsabilidades, interfaces e interacciones de alto nivel. Es la piedra angular de la mayoría de las DAs y la primera parte que suelen leer todos las partes interesadas. Es además el que orienta la forma de otros puntos de vista, tales como el informativo, el concurrente, el de despliegue y otros. Tiene además un impacto muy significativo en las propiedades de calidad del sistema, tales como la mantenibilidad, la seguridad y el rendimiento.
- Punto de vista informacional (o de datos).- Describe la forma en la que el sistema almacena, manipula, gestiona y distribuye la información. Desarrolla una vista completa,

²⁷Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*, parte III.

pero sólo a alto nivel, de la estructura estática de datos y el flujo de la información. Su objetivo es dar respuesta a preguntas fundamentales sobre el contexto, la estructura, la propiedad, la latencia, las referencias y la migración de información.

- Punto de vista concurrente.- Describe la estructura de concurrencia del sistema y mapea los elementos funcionales a unidades concurrentes para identificar de forma clara qué partes del sistema se ejecutan de forma concurrente y cómo se coordinan y controlan. Para ellos es necesario crear modelos que muestren el proceso y las estructuras de hebra que usará el sistema, así como los mecanismos de comunicación entre procesos para coordinar estas operaciones.
- Punto de vista de desarrollo.- Describe la arquitectura del proceso de desarrollo software. Las vistas de desarrollo comunican los aspectos de la arquitectura que necesitan las partes interesadas implicadas en la construcción, prueba, mantenimiento y mejora del sistema.
- Punto de vista de despliegue.- Describe el entorno en el que el sistema será utilizado y las dependencias que tiene el sistema con éste. Incluye el entorno hardware que requiere el sistema (nodos de procesamiento, conexiones de red, capacidades de almacenamiento en disco), los requisitos técnicos de cada elemento externo y el mapeo entre los elementos software y el entorno de ejecución que lo ejecutará.
- Punto de vista operacional.- Describe cómo el sistema será utilizado, administrado y mantenido una vez en fase de explotación. Entre las tareas más significantes que debe describir están las de instalación, gestión y uso del sistema. El objetivo de este punto de vista es el de identificar las estrategias de todo el sistema para responder a las inquietudes que las partes interesadas puedan tener sobre la operabilidad del sistema e identificar soluciones que den respuesta.

El punto de vista contextual describe las relaciones, dependencias e interacciones entre el sistema y su entorno (las personas, los sistemas y las entidades externas con las que interactúa).

Los puntos de vista funcional, informacional y concurrente caracterizan la organización fundamental del sistema.

Los puntos de vista de despliegue y operacional definen el sistema una vez puesto en explotación.

El punto de vista de desarrollo se usa para gestionar la creación del sistema.

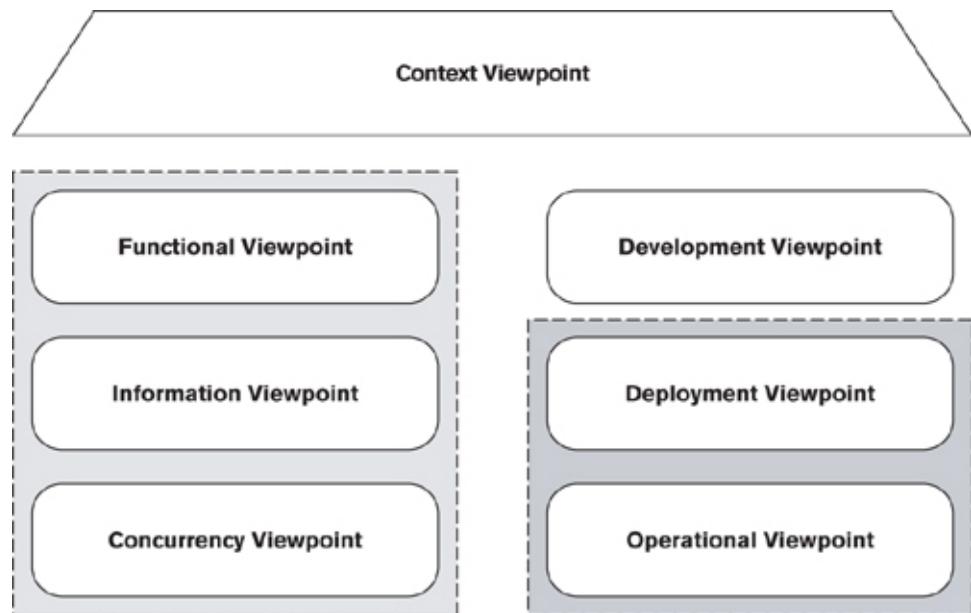


Figura 3.15: Agrupamiento de puntos de vista propuesto por Rozansky. [Fuente:²⁸]

Este catálogo no está cerrado. No hay que aplicar todos los puntos de vista, sino que dependerá de cada sistema concreto. Además pueden surgir nuevos puntos de vista.

Un primer paso para empezar a realizar una DA es entender la arquitectura del sistema, las habilidades y la experiencia de las partes interesadas, el tiempo disponible para el desarrollo, etc. para después seleccionar puntos de vista y vistas.

3.6.5. Plantilla para definir un punto de vista

Rozansky propone una plantilla (Tabla 3.2) para definir cómo aplicar un punto de vista en un proyecto software concreto.

²⁹Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition* (Addison-Wesley Professional, 2011), <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

Detalle	Descripción
Inquietudes	Aquellas que preocupan a las partes interesadas, identificando a aquéllos con más interés en este punto de vista
Modelos	Los modelos más importantes que se usarán para representar las distintas vistas, junto con las notaciones usadas y las actividades a realizar para construirlos
Problemas y errores comunes	Aquélllo que deba ser evitado, junto con técnicas a usar para reducir el riesgo
Lista de comprobación	Todas las cuestiones que no deban olvidársenos al desarrollar el punto de vista y cuándo deben revisarse para ayudar al buen desarrollo (correcto, completo, preciso)

Tabla 3.2: Plantilla para describir la aplicación de un punto de vista en un proyecto concreto.
[Fuente:²⁹]

3.6.6. Descripción detallada de un punto de vista: el punto de vista contextual

Veremos aquí una descripción detallada de cómo aplicar este punto de vista con algunos ejemplos.³⁰

Este punto de vista debería ser el primero en ser descrito aunque muchas veces no se incluye en la DA porque se supone que se conoce, y se aborda en primer lugar el punto de vista funcional, quizás añadiendo sólo un mero *diagrama de contexto*. Es muy importante describirlo de forma detallada porque puede ser necesario para cumplir algunos requerimientos del sistema y porque en el resto de puntos de vista podemos necesitar referirnos a los distintos elementos del sistema externo.

Para la descripción, adaptaremos la plantilla para puntos de vista (Tabla 3.2) a este punto de vista concreto, tal y como aparece en la Tabla 3.3.

Inquietudes Las inquietudes más importantes desde este punto de vista son:

- Ámbito del sistema y responsabilidades.- Considera cuáles son las responsabilidades principales del sistema. Debe ser breve y fácil de entender por todas las partes interesadas (no debe confundirse con la definición completa de los requisitos del sistema). Generalmente el ámbito viene ya impuesto, pero si no lo está, debemos definirlo a partir de consultar a las partes interesadas en el sistema. A veces se puede querer dejar de forma explícita lo que el sistema excluye.

³⁰Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*, cap. 16.

³¹Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition* (Addison-Wesley Professional, 2011), <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

Detalle	Descripción
Definición	Describe las relaciones, dependencias e interacciones entre el sistema y su entorno
Inquietudes	Ámbito del sistema y responsabilidades; identificación de las entidades externas y servicios y datos usados; naturaleza y características de las entidades externas; identificación y responsabilidades de las interfaces externas; naturaleza y características de las interfaces externas; otras interdependencias externas; impacto del sistema en su entorno; y completitud, consistencia y coherencia global
Modelos	Modelo de contexto y escenarios de interacción
Problemas y errores comunes	Entidades externas incorrectas u omitidas; dependencias implícitas omitidas; descripción de interfaces imprecisas u omitidas; contexto o ámbito implícito o asumido; interacciones especialmente complicadas; abuso de jerga
Partes interesadas	Todos las partes interesadas, pero especialmente los compradores, usuarios y desarrolladores
Aplicabilidad	Todos los sistemas

Tabla 3.3: Plantilla de descripción del punto de vista contextual. [Fuente:³¹]

EJEMPLO 3.6.5. Comercio electrónico al por menor

Un ejemplo de las responsabilidades que deben incluirse dentro del ámbito del sistema son:

- Presentar el catálogo de productos, con imágenes y descripción de cada producto
- Proporcionar una herramienta de búsqueda flexible (por nombre de producto, tipo, palabra clave, tamaño, etc.)
- Aceptar pedidos de productos
- Aceptar pago con tarjeta de crédito (con aprobación asíncrona y notificación al cliente)
- Proporcionar interfaces de conexión automática con un sistema de soporte para la entrega del producto

Como ejemplos de capacidades excluidas en una primera versión del sistema, tenemos:

- La capacidad para modificar o cancelar pedidos (sólo se podrá hacer por teléfono)
- La posibilidad de pagar de otra forma distinta a usar tarjeta de crédito
- Muestra del stock y posibilidad de reservar hasta disponibilidad

- Identificación de las entidades externas, servicios y datos usados.- Una entidad externa o actor es un sistema, organización o persona con el que interactúa de alguna forma nuestro sistema (ofreciendo/consumiendo servicios o datos a/de nuestro sistema). Algunos ejemplos son:
 - Otro sistema dentro de la empresa/institución que crea el sistema a modelar (se les suele referir como *sistemas internos* o *sistemas propios*)
 - Otro sistema que forma parte de una empresa/institución distinta (nos referimos a ellos como *sistemas externos*)
 - Una pasarela u otro componente software que oculta a otros sistemas (externos o internos)
 - Un almacén de datos externo al sistema (por ejemplo una base de datos compartida o un almacén de datos)
 - Un periférico u otro dispositivo físico externo al sistema (como servidores de mensajería compartida o robots de búsqueda)
 - Un usuario, un tipo de usuario u otra persona o rol tales como operadores o personal de apoyo
- Naturaleza y características de las entidades externas.- Pueden afectar de forma significativa a nuestro sistema. Algunos ejemplos son la estabilidad y disponibilidad de una entidad externa, su rendimiento, la localización física, la calidad de los datos, etc. En definitiva, criterios de calidad externos de esas entidades (no internos a ellas), es decir, que afecten al exterior, y por tanto a nuestro sistema. También podemos necesitar considerar las entidades externas que no son sistemas, como por ejemplo si un usuario habla el mismo lenguaje natural de nuestro sistema o si la pasarela para usar un fax compartido tiene características de funcionamiento que deban tenerse en cuenta.

EJEMPLO 3.6.6. Sistema de reserva de viajes

Estos sistemas intercambian información con otros similares de cualquier parte del mundo. Algunos de ellos, por ejemplo los situados en lugares exóticos, pueden tener interrupciones en su disponibilidad debido a diferencias horarias o mayor índice de fallos, lo cual puede hacer que nuestro cliente pierda su reserva, algo enormemente dañino.

Para evitarlo, las interfaces de los sistemas de reserva de viajes tienen que ser muy bien diseñadas, por ejemplo permitiendo *idempotencia* (repetir la misma operación un número determinado de veces sin que dé error), grabar los intentos en una base de datos y notificarlos a los operadores del sistema o permitir continuar transacciones o transferencias de datos en el punto en el que dio error en vez de empezar desde el principio.

- Identidad y responsabilidades de las interfaces externas. Hay que identificarlas todas, entendiendo que cada una de ellas puede cubrir alguno de los siguientes servicios:

- Proveedor/consumidor de datos, identificando los contenidos, ámbito y significado de los datos transferidos
- Proveedor/consumidor de servicios, identificando la semántica de la petición (incluyendo parámetros), las acciones a realizar para satisfacer la petición, los datos a devolver, acciones ante excepciones así como errores, estados, etc. a devolver
- Proveedor/consumidor de eventos, junto con la identificación de dichos eventos, su significado, contenido, volumen y tiempos de ocurrencia probables
- Naturaleza y características de las interfaces externas.- Puede haber una gran diferencia entre la calidad de las conexiones con las entidades externas (las interfaces externas) y los sistemas en sí, pudiendo la interfaz ser el cuello de botella o factor restrictivo en la interacción, por ejemplo cuando la conexión es de bajo ancho de banda y poco fiable pero el sistema con el que nos conecta es altamente resiliente. Algunos ejemplos de características de las interfaces incluyen:
 - Volúmenes esperados (número de peticiones o transferencias, tamaño de los datos, fluctuaciones por temporadas, crecimiento esperado con el tiempo)
 - Si las interacciones están programadas (para momentos predefinidos), ocurren como respuesta a eventos o son ad hoc
 - Si las interacciones están completamente automatizadas, o completamente manuales (como cuando el usuario guarda un fichero o envía un email) o algo intermedio
 - Si las interacciones son transaccionales, i.e. tienen que realizarse por completo
 - Estado crítico y exigencias de tiempo, como por ejemplo cuando se requiere que una interacción particular se termine antes del final del día para poder ser registrada en el sistema contable
 - Si las interacciones son de proceso por lotes (grandes conjuntos de datos son tratados como una unidad), basada en mensajes o de emisión secuencial (streaming)
 - El nivel de seguridad requerido (autenticación, autorización, confidencialidad, etc.)
 - El nivel de servicio que puede esperarse de la interfaz (en términos de tiempo de respuesta, latencia, escalabilidad, disponibilidad, etc.)
 - La naturaleza técnica de la interfaz y los protocolos usados (estándar abierto o propietario)
 - Formatos de datos y de ficheros
- Otras interdependencias externas.- A veces existen otro tipo de dependencias que no son de datos, de funciones o de eventos, desde o hacia en sistema a implementar, y que pueden ser difíciles de identificar. En esta inquietud debemos identificar la naturaleza de estas dependencias y su impacto en la arquitectura del sistema (qué funcionalidades deben añadirse a nuestro sistema para poder tener en cuenta estas dependencias)

EJEMPLO 3.6.7. Sistema de venta electrónica al por menor

Como ejemplo, en la Figura 3.16 se muestra un sistema de comercio electrónico (e-Commerce System) de venta al por menor, que interactúa con varias entidades software externas^a. Este es un ejemplo donde se muestran interdependencias externas entre los sistemas de reparto (Fulfillment System) y de cuentas de cliente (Customer Accounts System). El sistema de reparto tiene su propia lista de direcciones de reparto verificadas para cada cliente y rechazará cualquier petición que se haga a una dirección que no tenga registrada. Sin embargo, esta lista es una copia de la lista del sistema de cuentas de cliente. Cuando un cliente hace un pedido y además actualiza la dirección de envío, el sistema debe tener en cuenta esta dependencia, así como el tiempo para que la lista de direcciones se actualice en el sistema de reparto. Si no, los pedidos podrían ser rechazados porque el sistema de reparto aún no tenga esa dirección registrada.

El impacto que esta dependencia puede tener en la arquitectura es el de permitir el reenvío de la petición al sistema de reparto si ha sido rechazada la anterior, o retrasar las peticiones que conlleven actualización de la dirección de reparto para dar tiempo a que se actualicen las direcciones en el sistema de reparto.

^aEn este diagrama, se utiliza el estereotipo «system» para reflejar el sistema y el resto de cuadros, que aparecen sin estereotipo, significa que son entidades externas software.

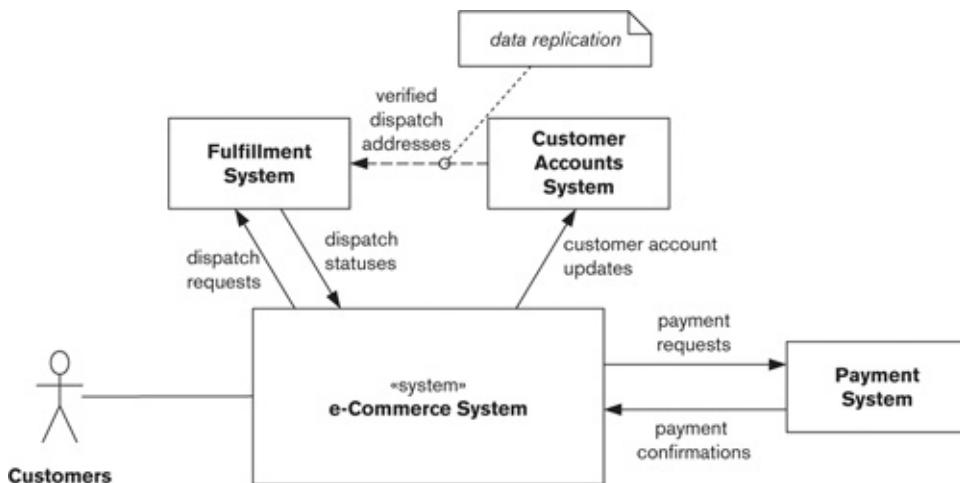


Figura 3.16: Un diagrama de contexto elaborado con UML. [Fuente:³²]

- Impacto del sistema en su entorno.- Se trata del impacto del sistema ya en explotación, tanto dentro de la organización como a nivel externo, e incluye aspectos como los siguientes:
 - Cualquier sistema que dependa del nuestro y que pueda requerir cambios a nivel funcional, de interfaces o mejoras en el rendimiento o en la seguridad

- Cualquier sistema que deba dejar de utilizarse cuando el nuestro entre en explotación
- Cualquier dato que deba migrarse a nuestro sistema
- Completitud, consistencia y coherencia global.- A menudo nuestro sistema será parte de un sistema mucho mayor, incluso compartido entre varias organizaciones mediante redes públicas o privadas. Este “paisaje global” puede llegar a ser muy complejo y difícil de entender.

Una preocupación (inquietud) de los usuarios particulares de nuestro sistema es que la solución global les provea de la funcionalidad que necesitan independientemente de qué sistema concreto implemente cada función o gestione cada conjunto de datos.

EJEMPLO 3.6.8. Sistema de venta electrónica al por menor

En los primeros sistemas de comercio electrónico se ponía todo el interés en proporcionar catálogos de productos de gran calidad para hacer los sitios atractivos a los compradores y ser competitivos. Sin embargo, se descuidaba los procesos de pago, reparto o tratamiento de excepciones. Esto finalmente les daba mala reputación en el servicio al cliente, incluso llegando a la quiebra.

Es cierto que esta inquietud es más responsabilidad del arquitecto de la empresa que del arquitecto de la aplicación, pero si se tiene en cuenta por el arquitecto de la aplicación puede aumentar la probabilidad de éxito de forma significativa. Es mucho más apreciable por los usuarios una aplicación consistente y coherente que una fragmentada e inconsistente.

Al menos se debe garantizar que los procesos de negocio más importantes tengan una cobertura adecuada, sea por sistemas manuales o automatizados. Del mismo modo, todos los datos que requieran los procesos deben almacenarse en algún lugar (por nuestro sistema o de forma externa) y ser accesibles a todos aquellos sistemas que los necesiten.

- Inquietudes para cada interesado en el sistema.- En la Tabla 3.4 se muestran las inquietudes que cada una de las partes interesadas en el sistema puede tener desde el punto de vista contextual.

Modelos: (1) **El modelo de contexto** El modelo más importante y muchas veces el único es el *modelo de contexto*, que presenta una imagen global del sistema completo y su relación con el exterior, y que suele incluir los siguientes tipos de elementos:

- El sistema en sí

³³Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*, Second Edition (Addison-Wesley Professional, 2011), Cap. 16, <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

Tipo de interesado	Inquietudes
Compradores (clientes)	Ámbito y responsabilidades del sistema; identificación de las entidades externas y servicios y datos usados; impacto de los sistemas en su entorno
Asesores	Todas las inquietudes
Comunicadores	Ámbito y responsabilidades del sistema; identificación y responsabilidades de las entidades externas; identidad y responsabilidades de las interfaces externas
Desarrolladores	Todas las inquietudes
Ingenieros de producción	Naturaleza y características de las interfaces externas; impacto del sistema en su entorno
Administradores del sistema	Todas las inquietudes
Equipo de prueba	Todas las inquietudes
Usuarios	Ámbito y responsabilidades del sistema; identificación de las entidades externas y servicios y datos usados; completitud, consistencia y coherencia global

Tabla 3.4: Inquietudes de las distintas partes interesadas desde el punto de vista contextual.
[Fuente:³³]

- Las entidades externas (tanto sistemas propios como sistemas externos)
- Las interfaces

Notación Debe contemplarse además la notación que será usada, siendo una de las más comunes [UML](#) (que no tiene diagrama de contexto como tal pero que puede ser creado a partir de un diagrama de casos de uso o de un diagrama de clases). Se representará cada componente de la siguiente forma:

- El sistema en sí como un componente y usando estereotipos.
- Los sistemas externos pueden representarse como componentes o actores [UML](#), usando el estereotipo «external».
- Las entidades externas que representan a usuarios que interactúan con el sistema serán representadas siempre como actores [UML](#).
- Las interfaces entre las entidades externas y el sistema pueden representarse en [UML](#) como flujos de información, dependencias o asociaciones con navegabilidad incluida (flechas vs líneas).

Puede verse un ejemplo en la Figura 3.17. Obsérvese que falta la navegabilidad en las líneas.

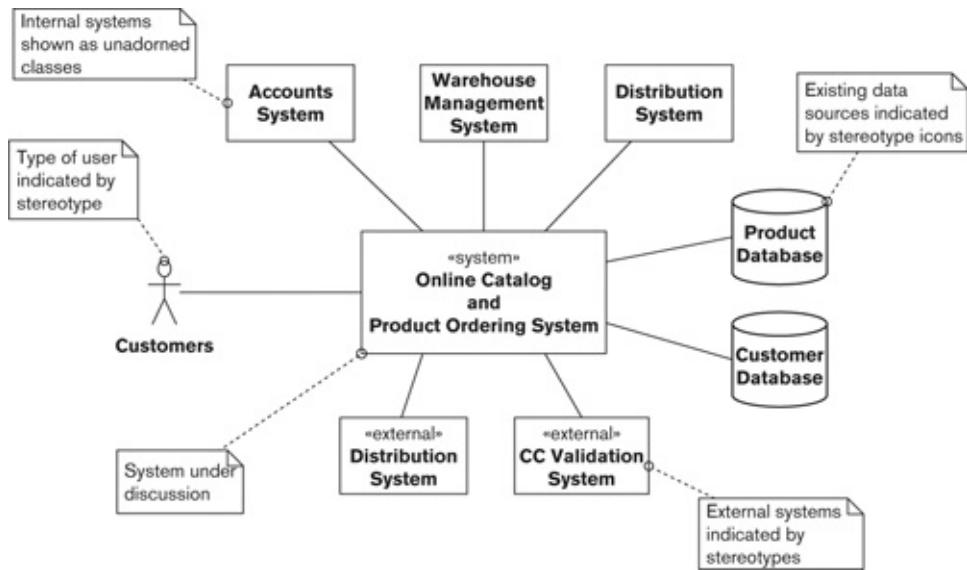


Figura 3.17: Un diagrama de contexto elaborado con **UML**. [Fuente:^{34]}

Puesto que **UML** no proporciona un soporte específico ni potente para los diagramas de contexto, otra posibilidad es usar diagramas informales con notación de *cuadros y líneas*, que puede ser más simple. Puede verse un ejemplo en la Figura 3.18.

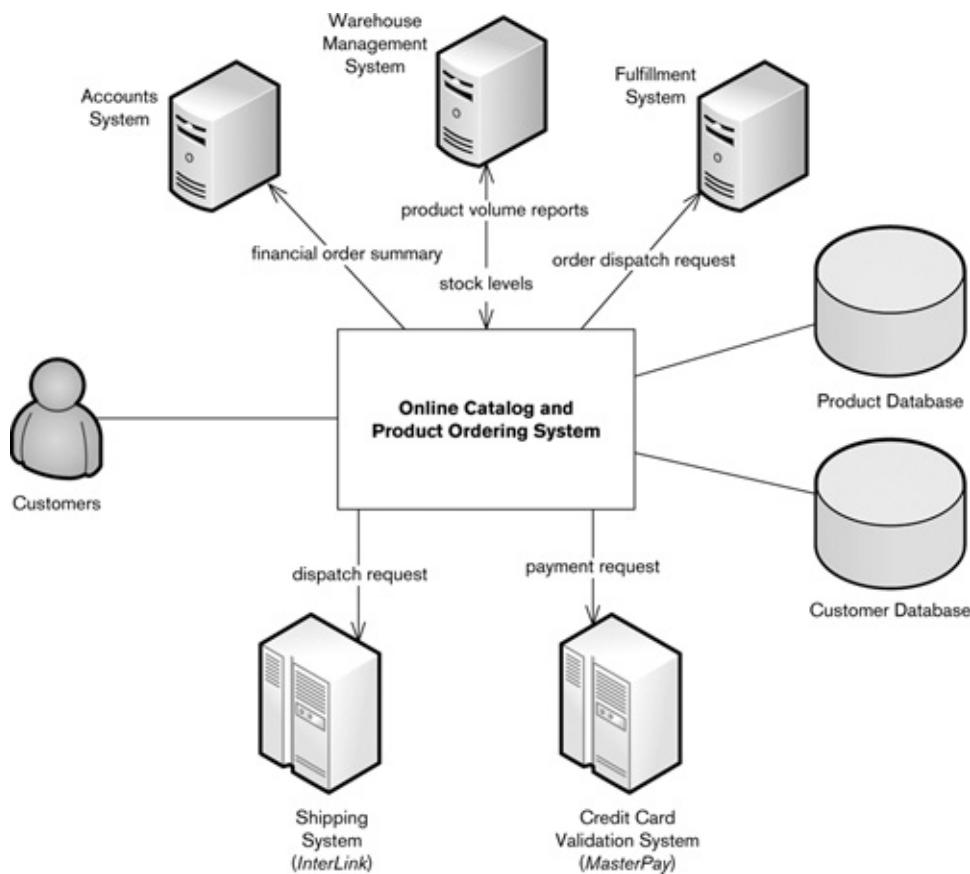


Figura 3.18: Un diagrama de contexto elaborado de forma libre con *cuadros y líneas*. [Fuente:³⁵]

La ventaja es que puede ser mucho más expresivo que UML y más fácil de entender por parte de algunos de las partes interesadas en el sistema. El problema es que obliga a explicar la notación.

Actividades Para realizar el diagrama de contexto hay que llevar a cabo las siguientes actividades previas, que deben formar parte del documento elaborado para describir la aplicación de este punto de vista en el sistema a desarrollar:

- Revisar los objetivos del sistema
- Revisar los requisitos funcionales claves
- Identificar las entidades externas
- Definir las responsabilidades de las entidades externas
- Identificar las interfaces entre el sistema y cada entidad externa
- Identificar y validar las definiciones de las interfaces

- Hacer un seguimiento del flujo de control e información entre el sistema y las entidades externas y añadir las nuevas interfaces que surjan
- Si se van a describir escenarios o casos de uso concretos, recorrerlos para validar el modelo y añadir cualquier entidad externa o interfaz que sea necesaria

Modelos: (2) Escenarios de interacción

DEFINICIÓN 3.6.3: Escenario de interacción

Un *escenario de interacción* representa a dos o más participantes (generalmente el sistema y una o más entidades externas) y una secuencia de interacciones entre ellas, siendo la interacción un flujo de información y/o una petición para realizar una acción. Todas las interacciones del escenario están encaminadas a cumplir un propósito o función concretos.

A menudo es útil modelar las interacciones que se esperan entre el sistema a desarrollar y entidades externas, de forma más detallada que lo que se muestra en el diagrama de contexto, para ayudar a identificar requisitos y restricciones implícitas (por ejemplo, ordenación, o restricciones de volumen o de tiempo) y ayudar a hacer una validación más detallada.

Si bien es poco probable que se tenga tiempo para modelar todos los escenarios en los que participará nuestro sistema, puede ser útil modelar algunos de los más complicados, polémicos o menos entendidos, especialmente cuando el uso del sistema no está claro o hay desacuerdo entre las partes interesadas.

DEFINICIÓN 3.6.4: Escenario arquitectónico

Un *escenario arquitectónico* es un escenario de interacción con una descripción bien definida de una interacción entre una entidad externa y el sistema. Define el evento que dispara el escenario, la interacción que inicia la entidad externa y la respuesta que se espera del sistema.

El arquitecto debe en todo momento ser capaz de asignar prioridades de forma correcta en caso de necesidades de distintas partes interesadas que sean incompatibles entre sí. A menudo es posible que el arquitecto se olvide de algunas prioridades del sistema impuestas por algunas partes interesadas y se deje llevar por sus propias preferencias personales. El uso de escenarios en la arquitectura, que permiten modelar algunas interacciones entre el sistema y entidades externas que sean especialmente clarificadoras, permitirá poder comprobar continuamente que las ideas que desarrolle funcionen de verdad en la práctica.

Algunos ejemplos de escenarios, para esta vista o para otra vista o perspectiva, son:

- Un conjunto particular de interacciones con sus usuarios a las que el sistema debe poder responder

- El procesamiento que debe realizarse automáticamente en un momento determinado, como a fin de mes
- Una situación particular de carga máxima que podría ocurrir (escenario de calidad)
- Una demanda que un regulador externo pueda hacer de un sistema
- Cómo debe responder el sistema a un tipo particular de fallos
- Un cambio que un encargado de mantenimiento podría necesitar hacer en el sistema
- Cualquier otra situación a la que el diseño del sistema deba poder hacer frente

Para hacer un uso efectivo de los escenarios se pueden seguir las siguientes pautas de actuación antes de modelar los escenarios concretos:

- Enfocarse en un conjunto concreto de escenarios, contando con las partes interesadas para priorizar cuáles son los más importantes para guiar la toma de decisiones.
- Usar escenarios distintos, evitando una tendencia a modelar escenarios parecidos que reducen mucho su utilidad.
- Incluir el uso de escenarios que tengan en cuenta criterios de calidad (ver las perspectivas y su relación con los puntos de vista en Sección 3.6.8).
- Incluir el uso de escenarios fallidos, como los que consideran que haya problemas de falta de información, sobrecarga, fallos de seguridad, etc., interesantes en especial para ayudar a garantizar los criterios de calidad.
- Involucrar lo más posible a las partes interesadas, que pueden ralentizar el proceso de descripción arquitectónica al proporcionar numerosa información, pero que lo van a enriquecer y hacerlo más real al ser capaces de ver aspectos y prioridades que el arquitecto puede desconocer por completo. Son además ellos los que deben priorizar los distintos aspectos del sistema. Esta pauta es muy importante y de ella depende enormemente el éxito del producto. Algunos autores recomiendan organizar *talleres colaborativos* para la construcción de los escenarios, de forma que el grupo tienda a aunar fuerzas y criterios. Para ello es muy importante que el coordinador haga una buena tarea como facilitador, manteniéndose neutral en todo momento.³⁶

DEFINICIÓN 3.6.5: Escenarios funcionales

Responden al qué, se relacionan con los requisitos funcionales y se suelen definir como una secuencia de eventos externos (derivados generalmente de un caso de uso) a los que el sistema debe responder de una forma particular.

³⁶Ellen Gottesdiener, «Running a use case/scenario workshop», en *Scenarios, stories, use cases through the systems development life-cycle*, ed. por Ian F. Alexander y Maide Mein (Wiley, 2004), <https://www.oreilly.com/library/view/scenarios-stories-use/9780470861943/>.

Los escenarios arquitectónicos que se usan en el punto de vista de contexto son escenarios funcionales. Otros tipos de escenarios (los escenarios de calidad del sistema) son utilizados para describir inquietudes transversales o perspectivas que atienden a requisitos no funcionales o de calidad (Sección 3.6.8).

Para describirlos suele proporcionarse la siguiente información:

1. Descripción general: una breve descripción de lo que el escenario debe ilustrar.
2. Estado del sistema: el estado del sistema antes de que ocurra el escenario (si es significativo). Esto suele ser una explicación de cualquier información que ya debería estar almacenada en el sistema para que el escenario sea significativo.
3. Entorno del sistema: cualquier observación importante sobre el entorno en el que se ejecuta el sistema, como la falta de disponibilidad de sistemas externos, el comportamiento particular de la infraestructura, las restricciones basadas en el tiempo, etc.
4. Estímulo externo: una definición de lo que hace que se dispare el escenario, como los datos que llegan a una interfaz, la entrada del usuario, el paso del tiempo o cualquier otro evento de importancia para el sistema.
5. Respuesta requerida del sistema: una explicación, desde el punto de vista de un observador externo, de cómo el sistema debe responder al escenario.

EJEMPLO 3.6.9. Escenario funcional en un sistema que sumariza los datos de entrada

Actualización estadística incremental

- Descripción general: cómo el sistema maneja un cambio en algunos de los datos base existentes
- Estado del sistema: ya existen estadísticas de resumen para el trimestre de ventas al que se refieren las estadísticas incrementales. Las bases de datos del sistema tienen suficiente espacio para hacer frente al procesamiento requerido para esta actualización
- Entorno del sistema: el entorno de implementación funciona normalmente, sin problemas
- Estímulo externo: una actualización de un subconjunto de las transacciones de ventas para el trimestre anterior llega a través de la interfaz externa *Datos de Carga Masiva (Bulk Data Load)*
- Respuesta requerida del sistema: los datos entrantes deberían activar automáticamente el procesamiento estadístico en segundo plano para actualizar las estadísticas resumidas del trimestre afectado para reflejar los datos actualizados de las transacciones de ventas. Las estadísticas de resumen anteriores deben permanecer disponibles hasta que las nuevas estén listas

Hay que tener en cuenta que los escenarios que identifican situaciones de fallo, a menudo no dan las respuestas de forma muy concreta ni robusta y pueden ser inaceptables. Pero al escribir el escenario ya se puede discutir con las partes interesadas en el sistema sobre el problema en cuestión y pensar en formas más concretas de afrontar las situaciones especiales.

Notación Para describir un escenario basta con usarse listas de interacción textual (a diferencia de cuando se definen los casos de uso completos), o bien diagramas de secuencia de [UML](#).

Actividades Descritas en el apartado [3.6.6](#).

Problemas y errores comunes Se incluyen los siguientes:

- Entidades externas ausentes o incorrectas
- Dependencias implícitas entre entidades externas no consideradas, y que pueden afectar al propio sistema
- Descripciones imprecisas o ausentes de una interfaz externa
- Nivel de detalle inapropiado
- Degeneración del entorno, o efectos progresivos de cambios no controlados que pueden no apreciar las partes interesadas
- Contexto o ámbito implícito o asumido
- Complicación de interacciones
- Abuso de jergas tecnológicas que pueden no entender las partes interesadas

Lista de comprobación Se propone la siguiente:

- ¿Has consultado con todos las partes interesadas quiénes están interesados en el punto de vista contextual (probablemente sean todos)?
- ¿Has identificado todas las entidades externas al sistema y sus responsabilidades más relevantes?
- ¿Comprendes bien la naturaleza de cada interfaz con cada entidad externa y está documentada en un nivel apropiado de detalle?
- ¿Has considerado las posibles dependencias entre las entidades externas con las que hay que interactuar? ¿Están documentadas estas dependencias implícitas en la [DA](#)?
- ¿Ilustra de forma adecuada el diagrama de contexto todas las interfaces entre el sistema y su entorno con las suficientes definiciones aclaratorias en el diagrama?

- ¿Han acordado formalmente todos las partes interesadas con los contenidos del modelo contextual? ¿Está documentado en algún lugar?
- ¿Está situado el modelo contextual bajo algún sistema formal de control de cambios?
- ¿Se sigue el proceso de control de cambios? ¿Se consulta a las partes interesadas para que den su consentimiento formal?
- ¿Se coloca en modelo contextual en algún lugar fácilmente accesible por todos, tal como una carpeta compartida?
- ¿Has identificado todas las capacidades o requerimientos básicos del sistema y están documentandos en el nivel de detalle apropiado?
- ¿Es la definición del ámbito consistente internamente?
- ¿Identifica el entorno cualquier posible restricción tecnológica como por ejemplo plataformas de uso exigidas?
- ¿Está el entorno especificado en un nivel apropiado de detalle, equilibrando brevedad con claridad y completitud?
- ¿Has explorado un conjunto de escenarios realistas de interacciones entre el sistema y actores externos?
- ¿Les parece claro el contexto, el ámbito y posibles implicaciones a otros equipos con los que debes interactuar?
- ¿Has comprobado si en el modelo contextual existe alguna información que parezca obvia y debería ser explícitamente descrita pero se ha omitido?
- Tienen los procesos de negocios más importantes una cobertura adecuada, tanto por los sistemas o los procesos manuales definidos?
- ¿Están todos los datos requeridos para los procesos de negocio principales almacenados en algún lugar, interna o eternamente?
- ¿Está formulada de forma coherente la solución global?

3.6.7. Descripción detallada de otro punto de vista: el punto de vista funcional

Veremos ahora una descripción detallada de cómo aplicar este otro punto de vista con algunos ejemplos.³⁷ El resto de puntos de vista están descritos con detalle en los capítulos 18 a 22 del libro de Rozansky.³⁸ Para la descripción, utilizaremos la plantilla para puntos de vista (Tabla 3.2), y la rellenaremos tal y como aparece en la Tabla 3.5.

³⁷Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*, cap. 17.

³⁸Rozanski.

Detalle	Descripción
Definición	Describe los elementos funcionales del sistema en ejecución, así como sus responsabilidades, interfaces e interacciones más importantes
Inquietudes	Capacidades funcionales, interfaces externas, estructura interna y filosofía del diseño funcional
Modelos	Modelo de la estructura funcional
Problemas y errores comunes	Interfaces pobremente definidas Responsabilidades no bien entendidas Infraestructuras modeladas como elementos funcionales Vista sobrecargada Diagramas sin definiciones de elementos Dificultades para reconciliar las necesidades de distintos partes interesadas Nivel de detalle erróneo <i>Elementos divinos</i> Demasiadas dependencias
Partes interesadas	Todos
Aplicabilidad	Todos los (sub)sistemas

Tabla 3.5: Descripción del punto de vista funcional. [Fuente:³⁹]

El punto de vista funcional de un sistema define los elementos arquitectónicos que implementan las funciones del sistema. Documenta la estructura funcional del sistema, incluyendo elementos funcionales clave, sus responsabilidades, sus interfaces y las interacciones entre ellos. Muestra cómo el sistema llevará a cabo las funciones que se requieren del mismo (requisitos funcionales).

Constituye la piedra angular de la mayoría de las DAs y probablemente la vista más fácil de entender por parte de las partes interesadas. Generalmente este punto de vista dirige además la definición de otros puntos de vista, sobre todo de los de información, concurrencia, desarrollo y despliegue. Generalmente se dedica mucho tiempo a refinar la estructura definida en este punto de vista.

Uno de los desafíos más importantes al describirlo es hacerlo al nivel adecuado de detalle, enfocándose en lo que es más significativo desde el punto de vista arquitectónico (lo que tiene mayor impacto en las partes interesadas) y dejando los detalles para los diseñadores⁴⁰. Se debe evitar documentar detalles de implementación física tales como servidores o infraestructura en este punto de vista, para no complicar los modelos y confundir a las partes interesadas, y dejar esos detalles para el punto de vista del despliegue.

³⁹Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition* (Addison-Wesley Professional, 2011), <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

⁴⁰A nivel de patrones, habría que usar patrones arquitectónicos, y dejar los patrones de diseño para un uso posterior por parte de los diseñadores.

Inquietudes Las inquietudes más importantes desde este punto de vista son:

- Capacidades funcionales.- Definen qué funciones debe llevar a cabo el sistema y cuáles no (de forma explícita o implícita) porque queden fuera del ámbito del sistema o porque se provean en cualquier otro sitio.
- Interfaces externas.- Son los datos, eventos y flujos de control entre nuestro sistema y otros sistemas. Pueden ir en los dos sentidos: desde o hacia nuestro sistema. Generalmente estas son las únicas interfaces que preocuparán a las partes interesadas no técnicas, aunque en esta vista hay que modelar también las interfaces entre los distintos módulos o componentes estructurales del sistema.

La definición de las interfaces debe incluir tanto la sintaxis (estructura de los datos, llamadas a procedimientos) como la semántica (significado o efecto).

- Estructura interna.- Normalmente puede haber varias formas de diseñar un sistema que cumpla con un conjunto de requisitos. Puede ser construido como una entidad monolítica o como colección de componentes de bajo acoplamiento; puede ser construido a partir de cierto número de paquetes estándar unidos mediante algún middleware apropiado o escrito desde cero; las funciones pueden cubrirse mediante servicios proporcionados por sistemas externos a través de una red o implementadas por la organización, etc. El desafío es saber elegir entre las distintas posibilidades una arquitectura que sea apropiada para satisfacer tanto los requisitos funcionales como los criterios de calidad (requisitos no funcionales).

La estructura interna del sistema se define mediante sus elementos internos, qué hace cada uno de ellos y cómo interactúan entre sí. Esta organización interna puede tener un gran impacto en las propiedades de calidad del sistema. Por ejemplo, en un sistema complejo que traspasa los límites de la empresa es generalmente más difícil garantizar la seguridad que si se trata de un sistema sencillo que se ejecuta en un par de ordenadores situados juntos.

- Filosofía de diseño funcional.- La mayoría de las partes interesadas en nuestro sistema sólo lo estarán en lo que el sistema hace y en sus interfaces con usuarios y otros sistemas. Sin embargo, algunos otros pueden estar interesados en saber cómo se ajusta la arquitectura a una serie de principios de diseño robusto. Otros, especialmente nuestros clientes, pueden querer de forma implícita que el sistema esté bien diseñado porque así será puesto en explotación de forma más rápida, barata y fácil.

Algunos ejemplos de características de diseño que pueden formar parte de una buena filosofía de diseño son:

- Coherencia: de forma que se ha hecho una descomposición en elementos correcta, para que interaccionen formando un todo.
- Cohesión: con todas las funciones relacionadas incluidas en un mismo elemento para realizar diseños menos proclives a errores.

- Consistencia: en cuanto a que los mecanismos y decisiones de diseño se aplican a través de toda la arquitectura facilitando el diseño y el mantenimiento.
- (Bajo) acoplamiento e interdependencia (separación de las inquietudes/funciones): de forma que haya pocas dependencias entre distintos elementos y facilite el diseño y el mantenimiento, aunque un mayor acoplamiento (sistemas monolíticos) puede aumentar la eficiencia.
- Extensibilidad⁴¹, flexibilidad funcional⁴² y generalidad, vs simplicidad: mayor extensibilidad, flexibilidad funcional y capacidad de generalización hace a los sistemas más difíciles de implementar e inefficientes pero más fáciles de evolucionar; la excesiva simplicidad los hace baratos y eficientes pero difíciles de evolucionar e incluso pueden no cumplir todos los requisitos.

Estas características de diseño tienen siempre un efecto positivo en algunas cualidades del sistema (criterios de calidad), tales como capacidad de evolución, flexibilidad y mantenibilidad, pero también en otras que parecen tener una relación no tan directa, tales como rendimiento, seguridad, escalabilidad, etc.

Los principios y los patrones arquitectónicos son buenas técnicas para hacer que el sistema cumpla con ciertas características de diseño y pueden guiar a los diseñadores a tomar decisiones de diseño que doten al sistema de las características que se consideren más importantes de garantizar.

- Inquietudes de cada interesado en el sistema.- En la Tabla 3.6 pueden verse las inquietudes más importantes para las distintas partes interesadas en el punto de vista funcional.

Modelos: Modelo de la estructura funcional Contiene normalmente los siguientes elementos:

- Elementos funcionales.- Un elemento funcional es una parte del sistema ejecutable (tiene sentido en el tiempo de ejecución) y bien definida (en oposición a lo que se define para el tiempo de diseño) con responsabilidades específicas y que presenta interfaces bien definidas que permiten su conexión con otros elementos. En el nivel de abstracción más simple es un módulo de código software, pero también puede ser un paquete de aplicación, o un almacén de datos o incluso todo un sistema completo.
- Interfaces.- Una interfaz es un mecanismo bien definido por el que un elemento puede acceder a las funciones de otro. Se define mediante entradas, salidas y semántica de cada operación ofrecida por el elemento, junto con la naturaleza de la interacción

⁴¹Facilidad para añadir nueva funcionalidad.

⁴²Facilidad para modificar en el futuro las funciones ya proporcionadas.

⁴³Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*, Second Edition (Addison-Wesley Professional, 2011), Cap. 17, <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

Tipo de interesado	Inquietudes
Clientes	Capacidades funcionales básicas e interfaces externas
Asesores	Todas las inquietudes
Comunicadores	Potencialmente todas las inquietudes, hasta cierto límite según el contexto
Desarrolladores	Estructura interna y cualidades de diseño básicas; capacidades funcionales e interfaces externas
Administradores del sistema	Filosofía de diseño funcional básica, interfaces externas y posiblemente la estructura interna
Equipo de prueba	Estructura interna y cualidades de diseño básicas; capacidades funcionales e interfaces externas
Usuarios	Capacidades funcionales básicas e interfaces externas

Tabla 3.6: Inquietudes de los distintos tipos de partes interesadas en el punto de vista funcional. [Fuente:⁴³]

que se requiere para invocar una operación concreta, por ejemplo llamada (síncrona) a procedimiento remoto (Remote Procedure Call, RPC), mensajería (invocación asíncrona), eventos, interrupciones, etc.

- Conectores.- Son las partes de la arquitectura que unen los distintos elementos para que puedan interactuar. Un conector define la interacción entre los elementos que lo usan y permite que la naturaleza de la interacción se considere aparte de la semántica de la operación que se invoca. La naturaleza de las interacciones entre elementos pueden ser fuertemente asociadas a la forma en la que los elementos se conectan.

Sin embargo, el nivel de especificación de los conectores depende del tipo de interfaz. Si se usa RPC, apenas habrá que decir nada, sólo indicar qué elementos están conectados entre sí. Si se usa una interfaz basada en mensajería (como por ejemplo *Representational State Transfer* (REST), que está basado en el protocolo HTTP, se puede definir un conector como un elemento específico que proporciona la posibilidad de permitir la interacción entre dos entidades a través de él. Por ejemplo, en un servicio web basado en REST (servicio RESTFul), un conector puede ser una API que permita conectar una entidad con otra, como un servicio REST, de forma segura, pasando la información en formato JSON o XML.

- Entidades externas.- Lo normal es que no tengan consideración desde el punto de vista funcional, sino desde el punto de vista de contexto y en el de concurrencia y el de despliegue (en estos últimos para especificar cómo se ejecuta en hebras o en procesos –concurrencia– y cómo se empaquetá –despliegue–). Todo lo relacionado con la infraestructura también debe ser considerado desde el punto de vista de despliegue en vez de desde el funcional.

Por ejemplo, puede hacerse referencia al uso de colas de mensajes como conectores

pero el agente concreto que proporciona las colas debe ser parte del punto de vista de despliegue y no del funcional.

Notación Para este punto de vista podemos usar distintas técnicas de representación de modelos:

- Diagrama de componentes UML.- La mayor ventaja de usar UML es que es ampliamente conocido y flexible. El diagrama principal para el punto de vista funcional es el diagrama de componentes, que muestra los elementos del sistema, las interfaces y las conexiones entre los elementos.

EJEMPLO 3.6.10. Sistema de vigilancia de la temperatura

La Figura 3.19 muestra la estructura funcional de un sistema de vigilancia de temperatura usando UML. El sistema está formado por dos componentes internos: el subsistema que obtiene la temperatura (Variable Capture), con la interfaz VariableReporting, y el subsistema que dispara la alarma (Alarm Initiator), con la interfaz LimitCondition. El primero de ellos interactúa con un sistema externo que muestra la temperatura (Temperature Monitor) y que invoca a la interfaz VariableReporting, de la cual se da más información: es una RPC en XML que usa el protocolo HTTP (por tanto se usa de forma asíncrona, de forma más parecida a las interfaces de tipo mensajería que a las RPCs) y que permite un máximo de 10 invocaciones simultáneas.

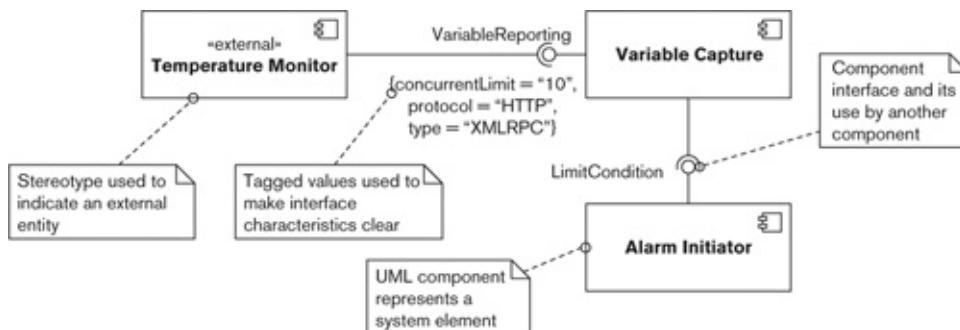


Figura 3.19: Ejemplo de una Estructura Funcional en UML. [Fuente: [44](#)]

Además del estereotipo «external» ya visto cuando se trató el punto de vista contextual, otro estereotipo usado para el punto de vista funcional es el «infrastructure», para indicar los elementos de la infraestructura del sistema que tenga una función diferenciada.

En el diagrama se ha optado por la representación de la interfaz mediante un pequeño ícono chupete (lollipop) que por una clase con el estereotipo «interface». Para dar más información sobre una interfaz, podemos usar valores etiquetados que nos digan de qué tipo es, el protocolo usado o el número de conexiones concurrentes permitidas.

Los conectores entre las interfaces se pueden representar como dependencias [UML](#) y flujos de información, como se describe en el siguiente ejemplo.

EJEMPLO 3.6.11. Tienda Web dentro de un entorno software ya existente en la empresa

La Figura 3.20 muestra un diagrama de componentes [UML](#) para describir la estructura funcional de un sistema sencillo. El sistema proporciona una tienda Web (Web Shop) que usan los clientes para compras a partir de un catálogo en línea y que será parte de un entorno software ya existente en la empresa.

El modelo muestra que el sistema interactúa con cinco entidades externas: los navegadores web de los tres tipos de usuario más importantes (personal de servicio al cliente, clientes y administrados del catálogo de productos), y dos sistemas externos (sistema de reparto e inventario). El sistema en sí está compuesto por cinco componentes principales, unidos por distintos tipos de conectores (incluyendo HTML sobre HTTP, mensajería por publicación/subscripción y una interfaz externa LU 6.2 (desarrollada por IBM).

Los clientes hacen el pedido a través de la tienda Web (Web Shop), que interactúa a su vez con el gestor de pedidos (Order Processor), el catálogo de productos (Product Catalog) y el sistema de información del cliente (Customer Information System). Los administradores del catálogo mantienen el catálogo a través de su interfaz basada en la Web (Catalog Management Interface) y los que forman parte del equipo de servicio al cliente mantienen la información del cliente (Customer Information System) mediante un programa cliente de interfaz dedicada (Customer Care Interface). Cuando se desea consultar el stock de un producto determinado, el catálogo de productos (Product Catalog) obtiene la información del inventario (Stock Inventory), que es un sistema que ya existe. Hay también cierta información en cuanto a la naturaleza de las interacciones entre componentes. Sabemos que pueden acceder al sistema simultáneamente hasta 1000 clientes, 80 personas del servicio al cliente y 15 administradores del catálogo. Además, la interacción entre el catálogo de productos y el inventario tiene lugar mediante un protocolo específico (que seguramente era un tecnología que ya existía por ser el inventario parte del software actual de la empresa). Podemos asumir según el ejemplo que la comunicación entre componentes que no ha sido descrita se lleva a cabo con algún procedimiento RPC (que será definido en algún otro lugar de la [DA](#) de forma clara).

Uno de los aspectos interesantes que se reflejan en este diagrama es la cantidad de deducciones no obvias que se pueden hacer a partir de él. Las responsabilidades de los componentes no están claras, tampoco los detalles de las interfaces ni los detalles de la interacción entre los distintos componentes. Esto nos hace entender la necesidad de añadir descripciones de texto y otros diagramas que modelen el sistema de forma complementaria, por ejemplo, las interacciones entre componentes puede mostrarse modelando escenarios del sistema.

- Otras notaciones formales de diseño.- [UML](#) no es la única notación de diseño bien definida adecuada para el desarrollo de software. Existen otras notaciones estructuradas más antiguas (como Yourdon, Jackson System Development y Object Modeling Technique de James Rumbaugh) que se han aplicado con éxito a problemas de desarrollo de software durante muchos años. El problema es que tienden a ser bastante débiles para describir los conceptos (como elementos a gran escala, interfaces, opciones de implementación, etc.) que son importantes para los arquitectos. Los métodos más antiguos tampoco se enseñan ni se usan ampliamente en la actualidad, por lo que puede ser difícil encontrar soporte de herramientas y carecen de la familiaridad general que tiene [UML](#) para la mayoría de las personas
- Lenguajes de descripción de arquitectura (Architecture description languages, ADLs): los lenguajes que admiten directamente los conceptos que interesan a los arquitectos de software se conocen generalmente como ADL. Se ha creado una gran cantidad de ADLs (incluidos Unicon, Wright, xADL, Darwin, C2 y AADL). El gran atractivo de los ADLs es que brindan soporte nativo para algunas de las cosas que necesitamos capturar y razonar en nuestros diseños arquitectónicos (como componentes y conectores). Sin embargo, casi todos los ADL se han desarrollado en el entorno de investigación y tienden a sufrir una serie de inconvenientes prácticos, incluida la falta de familiaridad de las partes interesadas en ellos, un alcance relativamente limitado (a menudo sólo permite que se representen los componentes y conectores), y una inevitable falta de soporte en herramientas maduras. Por estas razones, no se puede recomendar ninguno para uso cotidiano
- Diagramas de *cuadros y líneas*: muchos arquitectos utilizan un diagrama de estructura funcional basado en notación personalizada de cuadros y líneas. Dicho diagrama debe mostrar sólo los elementos funcionales y sus interfaces y debe vincular los elementos a las interfaces que usan con un dispositivo gráfico claro (generalmente una flecha, posiblemente con alguna anotación) que indica el uso de un conector. Al igual que con cualquier notación personalizada, se debe definir claramente el significado de la notación para evitar confusiones. La Figura 3.21 presenta el diagrama de *cuadros y líneas* equivalente al diagrama [UML](#) de la Figura 3.20. A veces es conveniente usar estos diagramas para vender las propiedades y beneficios del sistema a algunas partes interesadas y dejar los detalles técnicos para un diagrama [UML](#) que usen sólo algunas partes interesadas
- Bocetos: Permiten crear una sensación menos formal, introduciendo una notación ad hoc para representar cada uno de los aspectos de la vista que sean significativos para el sistema. A menudo es necesario para comunicar de manera efectiva aspectos esenciales de la vista a las partes interesadas no técnicas. El problema es que pueden conducir a una visión mal definida y confusión entre las partes interesadas. Al igual que con el diagrama de *cuadros y líneas*, puede evitarse esto utilizando un boceto como añadido a una notación de vista más formal (como [UML](#)) y utilizando diferentes anotaciones para diferentes grupos de partes interesadas

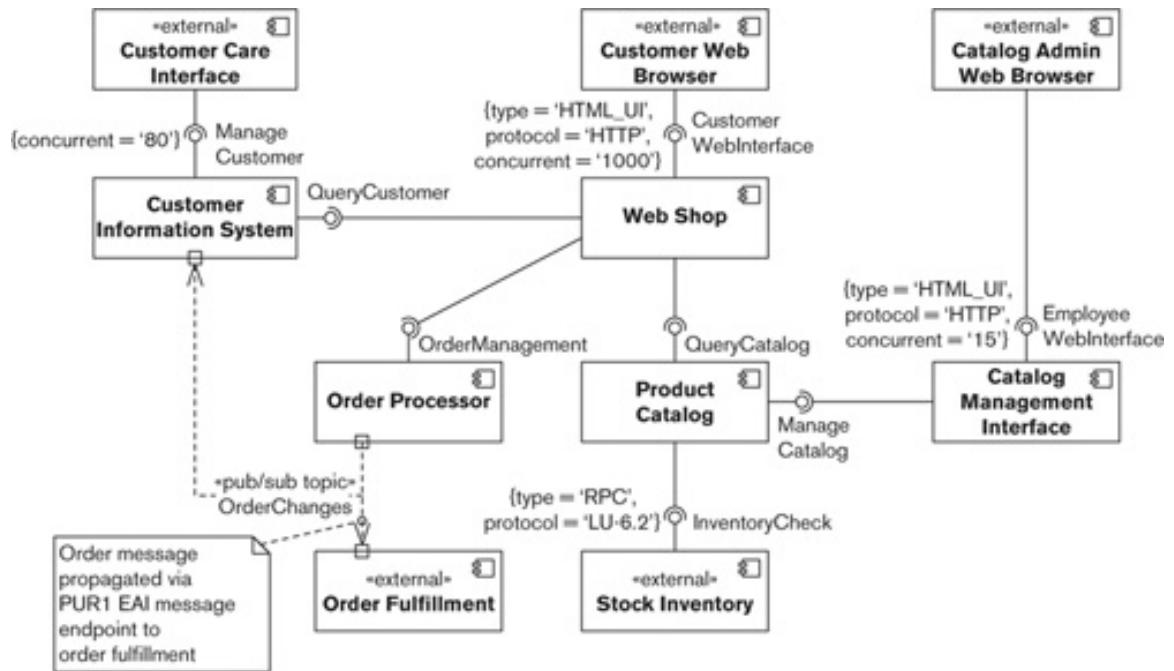


Figura 3.20: Ejemplo de diagrama de componentes UML para modelar una tienda Web.
[Fuente:⁴⁵]

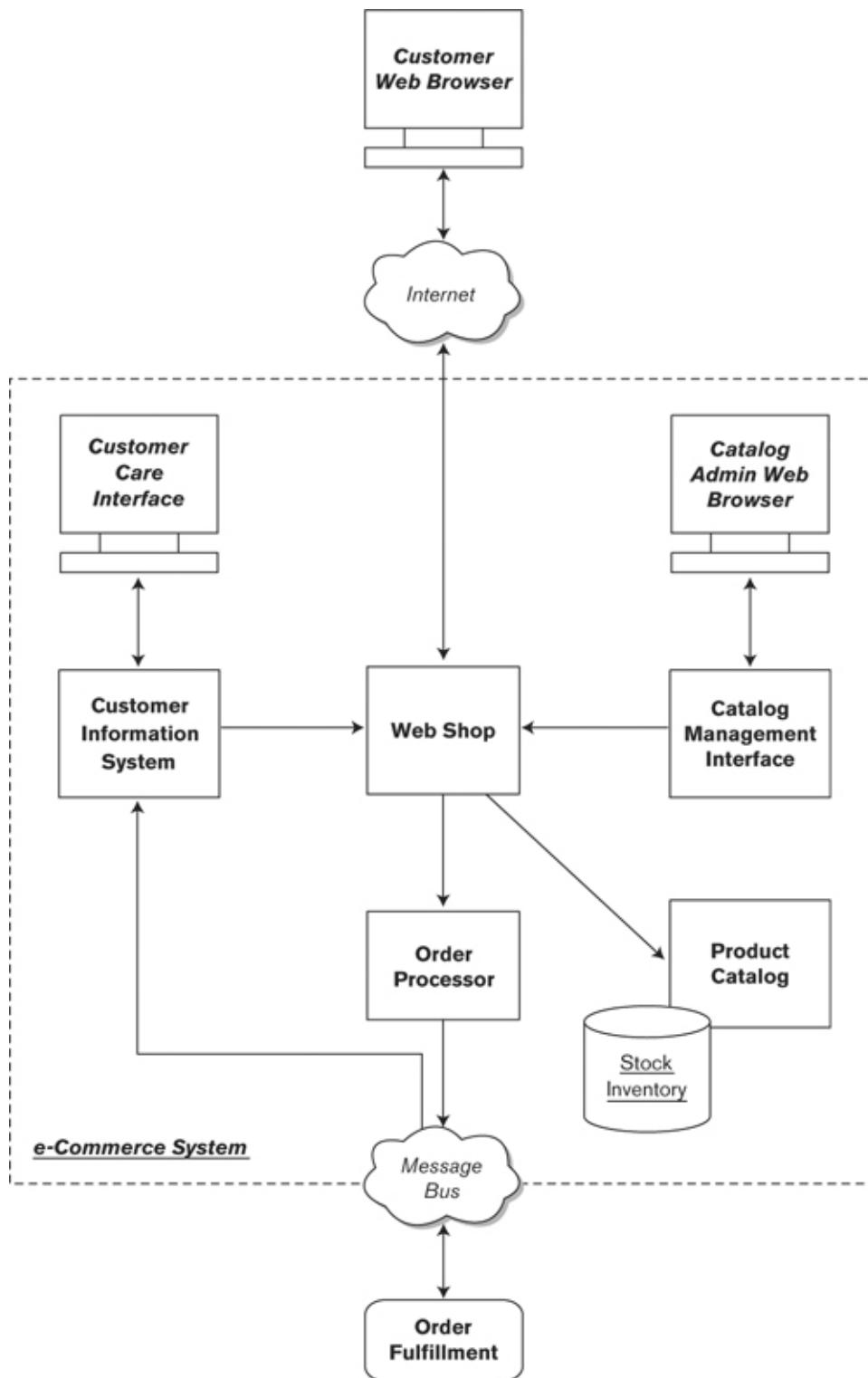


Figura 3.21: Ejemplo de diagrama de *cuadros y líneas* equivalente al diagrama de componentes UML para modelar una tienda Web que aparece en la Figura 3.20. Se ha usado una notación propia: las interfaces de usuario externas se representan por un ícono que simboliza el monitor del ordenador, y los sistemas externos de respaldo se representan por rectángulos con esquinas redondeadas. Los almacenes de datos se representan por un ícono que simboliza un tambor de discos y las interfaces funcionales (Internet, el bus de mensajes –Message Bus–) se representan mediante una nube. El ámbito del sistema incluye a todos los elementos dentro del rectángulo de línea discontinua. [Fuente: [46](#)]

Actividades

- Identificar los elementos.- Hay muchas formas de hacerlo, y el método correcto a utilizar depende del tipo de sistema y el enfoque de desarrollo de software que estamos utilizando. Por ejemplo, enfoques procedurales clásicos, orientación a objetos o enfoques basados en componentes, influyen en la identificación de componentes de diferentes maneras. De todos modos, en todos ellos pueden seguirse los siguientes pasos:
 1. Trabajar a partir de los requisitos funcionales, derivando responsabilidades clave a nivel del sistema
 2. Identificar los elementos funcionales que desempeñarán esas responsabilidades
 3. Evaluar el conjunto identificado con los criterios de diseño deseables
 4. Repetir la secuencia para refinar la estructura funcional hasta que juzgue que es sólida, usando uno o más métodos de refinamiento entre los siguientes:
 - Generalización: identifica algunas responsabilidades comunes en una serie de elementos e introduce una cantidad de elementos más generales que pueden reutilizarse en todo el sistema para realizar estas tareas. La generalización es particularmente importante como parte de una arquitectura de línea de producto o empresa más grande para permitir la reutilización de activos de software en una serie de productos o sistemas similares
 - Descomposición: divide un elemento grande y complejo en varios subelementos más pequeños. Para sistemas grandes, a menudo necesitamos dividir los elementos funcionales de nivel superior en elementos a nivel de subsistema más manejables para permitir su diseño y construcción
 - Composición: reemplazar varios elementos funcionales pequeños con un elemento más grande que incluye todas las funciones de los más pequeños. La composición se usa típicamente cuando se ha identificado un gran número de elementos funcionales pequeños pero similares. En tales casos, a menudo tiene sentido desde una perspectiva arquitectónica reemplazar los elementos más pequeños con un solo elemento grande que pueda factorizar la parte común entre los más pequeños y reducir la cantidad de interacciones que requiere el sistema
 - Replicación: repetir un elemento del sistema o una pieza de procesamiento. Un ejemplo es la validación de datos, donde se identifica un elemento de validación para los datos entrantes y luego los replica en varias interfaces externas del sistema. La replicación puede traer beneficios de rendimiento, pero se debe tener cuidado para mantener consistentes los componentes replicados⁴⁷

⁴⁷Salvo en sistemas que tengan restricciones muy cortas de tiempo de desarrollo y que no vayan a tener ningún mantenimiento, este método de refinamiento está muy desaconsejado porque el aumento de rendimiento se hace a costa de un enorme riesgo de inconsistencia en los componentes duplicados.

Si se está utilizando un estilo arquitectónico para guiar el proceso de diseño, el proceso es ligeramente diferente porque implicará crear una instanciación del estilo de modo que las responsabilidades a nivel del sistema se asignen a elementos del estilo. Esta actividad está estrechamente relacionada con el siguiente paso: asignar responsabilidades a los elementos.

- Asignar responsabilidades a los elementos.- Una vez que se han identificado los elementos candidatos, la siguiente actividad consiste en asignarles responsabilidades claras, es decir, la información gestionada por el elemento, los servicios que ofrece a otras partes del sistema y las actividades que inicia, si es que no se ha completado ya en el paso anterior

EJEMPLO 3.6.12. Responsabilidades de dos elementos de la aplicación de comercio electrónico de venta al por menor antes descrita

- Elemento: Tienda Web
 - Proporcionar a los clientes una interfaz HTML accesible desde el navegador Web
 - Gestionar todos los estados relacionados con la sesión de la interfaz del cliente
 - Interaccionar con otras partes del sistema para permitir que el cliente pueda ver el catálogo de productos y el stock, comprar y consultar información sobre su cuenta
- Elemento: Sistema de Información al Cliente
 - Gestionar toda la información persistente relacionada con los clientes del sistema
 - Proporcionar una interfaz sólo-consulta al cliente que le permita consultar la información a la que tenga acceso
 - Proporcionar una interfaz programable de gestión de información que pueda usarse para crear aplicaciones de gestión de información de clientes
 - Proporcionar una interfaz dirigida por eventos de manejo de mensajes que acepte las líneas de detalle de los pedidos hechos por los clientes y los cambios de estado realizados sobre dichos pedidos

- Diseñar las interfaces.- Debe incluir las operaciones que ofrece la interfaz; la entrada, salidas, condiciones previas y efectos de cada operación; y la naturaleza de la interfaz (mensajería, RPC, servicio web, etc.)

El enfoque de *Diseño por Contrato* desarrollado para OO por Bertrand Meyer es aquí también muy apropiado, de forma que las interfaces se especifican a través de

“contratos” que utilizan condiciones previas, condiciones posteriores e invariantes para definir con precisión el comportamiento y las relaciones de cada operación.

La notación apropiada para la definición de interfaces depende del tipo de interfaz y de quién necesita comprender esta información (considerando factores como la tecnología de implementación probable, los antecedentes del equipo de desarrollo y los tipos de interfaces que deben describirse). Las siguientes son algunas notaciones comunes de definición de interfaz:

- Lenguajes de programación.- Se pueden definir directamente mediante el uso de un lenguaje de programación para especificar la firma de la operación junto con texto o código para la semántica de la operación. Este enfoque es simple pero lo vincula con el estilo, los supuestos y las limitaciones del lenguaje de programación particular, lo cual no es lo ideal, especialmente si se utilizan múltiples tecnologías. Es el enfoque más apropiado para bibliotecas software u otros ejemplos donde se usa un solo lenguaje de programación para implementar todo el sistema
- Lenguajes de definición de interfaz (Interface Definition Language, IDL).- Desarrollados para admitir tecnología de sistemas distribuidos de lenguaje mixto. Existen IDLs para CORBA, para .NET, o para Web Services Description Language (WSDL), una tecnología XML para describir servicios web. Independientes de la tecnología de implementación, tienden a permitir instalaciones más simples que los lenguajes de programación, resultando más adecuados para definir interfaces arquitectónicas. Siempre que las partes interesadas puedan leerlos (o que se les enseñe a leerlos), los IDLs constituyen una buena opción para definir firmas de operaciones
- Enfoques orientados a datos.- Permite describir las interfaces únicamente en términos de mensajes que se intercambian. Como ejemplos se incluyen interfaces a las que se accede a través de sistemas de mensajería e interfaces definidas en términos de intercambio estructurado de documentos (por ejemplo, interfaces orientadas a documentos y basadas en servicios web con mensajes definidos mediante el esquema XML). Recomendado en interfaces basadas en eventos que se definen en términos del intercambio de eventos comerciales en lugar de la invocación de operaciones

Cualquiera que sea la notación que se use para describir las interfaces, no hay que olvidar que una interfaz es mucho más que una simple definición de cómo llamar a las operaciones. Desafortunadamente, ninguno de los enfoques que hemos descrito ofrece facilidades para definir la semántica de la interfaz, por lo que una definición clara de una interfaz implicará el uso de lenguaje natural o lenguajes especializados como Object Constraint Language (OCL) para lograr esto. Una definición de interfaz debe comunicar con precisión las condiciones previas y posteriores de cada operación y cómo se deben combinar las operaciones para realizar una función útil (preferiblemente con ejemplos). Cualquier reducción en la descripción puede causar problemas importantes al utilizarlas.

- Diseñar los conectores.- Los elementos de un sistema deben relacionarse entre sí para lograr los objetivos del sistema, de forma que al identificar las responsabilidades de un elemento aparece la necesidad de interactuar con otros para poder llevarlas a cabo. Las interacciones tienen lugar a través de conectores de algún tipo que vinculan los elementos que delegan responsabilidades con las interfaces ofrecidas por los elementos en los que éstas se delegan. A veces, el tipo de conector requerido es evidente (como una simple llamada a procedimiento), mientras que en otros casos deberá pensar detenidamente si necesita comunicación sincrónica o asincrónica, la resiliencia requerida del conector, la latencia aceptable de interacciones a través de él, etc. Para cada vía de comunicación entre elementos requerido por la arquitectura, debe agregarse un conector al modelo para admitirla (ya sea RPC, mensajería, transferencia de archivos u otros mecanismos)
- Comprobar la trazabilidad funcional.- La especificación de requisitos funcionales define una serie de funciones que el sistema debe proporcionar. Es necesario comprobar la trazabilidad para asegurarse de que la estructura funcional propuesta cumpla con todos los requisitos funcionales, de forma que no falten en la estructura ninguna función o haya alguna incompleta. Para hacerlo de manera formal se puede usar una tabla que cruce los requisitos funcionales con los elementos del modelo funcional encargados de implementarlos
- Recorrer escenarios comunes.- Puede ser extremadamente valioso y esclarecedor recorrer con algunas partes interesadas, los escenarios comunes de uso del sistema, utilizando el punto de vista funcional para mostrar cómo se comportará el sistema en cada caso. Será especialmente útil hacerlo con miembros del equipo de pruebas, el equipo de desarrollo y los administradores del sistema. Debe explicarse cómo interactuarían los elementos del sistema para implementar el escenario, de forma que puedan identificarse debilidades arquitectónicas, malentendidos, o elementos omitidos
- Analizar las interacciones.- Es útil analizar la estructura elegida desde el punto de vista del número de interacciones entre elementos tomadas durante escenarios de procesamiento comunes, para tratar de reducir las interacciones a un conjunto mínimo sin distorsionar la coherencia de los componentes funcionales (bajar el acoplamiento manteniendo la cohesión). Por lo general, es un paso importante hacia un sistema eficiente y confiable. A veces habrá que compensar la reducción de interacciones entre elementos para que no resulte en una estructura distorsionada con redundancia indeseable o partición de elementos inapropiada
- Analizar la flexibilidad.- Un sistema que funcione siempre está bajo presión para cambiar, por lo que la arquitectura debe ser flexible. Dentro de ella, la estructura funcional es uno de los principales factores que afectan a la flexibilidad de los sistemas de información. Es útil analizar algunos escenarios de “qué pasaría si” que revelen el impacto de posibles cambios futuros en el sistema. Un problema común en este punto es que los cambios implicados para aumentar la flexibilidad pueden entrar en conflicto

con los sugeridos por el análisis de las interacciones. Por lo tanto, es importante encontrar el equilibrio adecuado entre complejidad y flexibilidad

Problemas y errores comunes

Se incluyen los siguientes:

- Interfaces pobemente definidas.- Muchas veces se definen muy bien los elementos, sus responsabilidades y relaciones entre sí pero se descuidan las interfaces, lo que puede llegar a malas interpretaciones en los equipos de desarrollo, que pueden terminar en un sistema poco fiable. Las interfaces deben incluir las operaciones, su semántica y ejemplos cuando sea posible
- Responsabilidades mal entendidas.- Deben definirse formalmente todas las responsabilidades de cada elemento para que los equipos de desarrollo no olviden ninguna o las repitan en más de un elemento
- Infraestructura modelada como si fueran elementos funcionales.- La infraestructura se debe modelar en el punto de vista de despliegue; hacerlo en el funcional, salvo que sea importante para entender la vista, le añade complejidad innecesaria
- Vista sobrecargada.- El punto de vista funcional es la piedra angular de la DA, la vista central, pero no el conjunto de todas las vistas (por ejemplo, incluyendo elementos de los puntos de vista de despliegue, concurrente o de información. Si se optara por una vista compuesta de todos los puntos de vista, habría que especificarlo claramente

EJEMPLO 3.6.13. Diagrama de una vista funcional sobrecargada

La Figura 3.22 muestra un ejemplo de vista funcional sobrecargada. Además de UML se ha añadido notación ad hoc: las líneas discontinuas en los elementos dentro del cuadro del nodo servidor (“Server Node”). Es difícil entender su significado sin preguntar al arquitecto o leer documentación adicional: representan procesos independientes del Sistema Operativo, lo cual no debería formar parte del punto de vista funcional. Tampoco debería estar en este punto de vista nada relacionado con el despliegue ni los distintos ordenadores que participan o el software externo (no deberían por tanto aparecer los elementos del nodo servidor. Por otro lado, los elementos del nodo servidor son ambiguos y los propios desarrolladores o personal del equipo de prueba necesitarán más detalle (en los puntos de vista de despliegue, de información, concurrente, etc.) para poder implementar y probar la solución.

- Diagramas sin definiciones de elementos.- A menudo se representa la estructura funcional (relaciones entre los elementos) sin dar una definición clara y precisa de cada elemento que todas las partes interesadas puedan entender

- Dificultades para reconciliar las necesidades de distintas partes interesadas.- A menudo no todas las partes interesadas pueden entender un mismo diagrama. Puede proporcionarse un diagrama específico para las partes técnicas y otro para las no técnicas (las partes interesadas en el negocio), que puede ser una simplificación del primero y una notación más sencilla
- Nivel de detalle inadecuado.- Hay que evitar diagramas tan detallados que bajen a capas de diseño demasiado detallado (por ejemplo, bajar a más de un tercer nivel de detalle posiblemente sea demasiado), pero también evitar ser tan genéricos que no se entiendan las ideas ni tampoco garantizar criterios de calidad (por ejemplo, quedarse en un primer nivel de detalle en sistemas muy grandes)
- Evitar un elemento “divino”.- A menudo en el diseño OO existe el riesgo de distribuir mal las responsabilidades entre los objetos y dejar casi todo a un sólo objeto muy grande, el gestor. Este problema, llamado el problema del *objeto divino*, tiene un homólogo al hacer una DA, de forma que un solo elemento acumule la mayor parte de la responsabilidad del sistema, haciéndolo demasiado complejo y difícil de entender. Además caerá sobre él casi toda la responsabilidad de asegurar los distintos criterios de calidad (tales como rendimiento, escalabilidad y fiabilidad) y más difícil de garantizarlos dada su complejidad

EJEMPLO 3.6.14. Diagrama de componentes UML con un elemento divino

La Figura 3.23 muestra un diagrama de componentes UML que puede adolecer del problema del elemento divino. En concreto, el Gestor de Clientes (Customer Management) parece tener una enorme responsabilidad, con el resto de elementos interactuando con él.

- Demasiadas dependencias.- El problema opuesto al del elemento divino es el tener diagramas con muchas dependencias entre elementos, con una aspecto como de arañas luchando por el control. Cuando las interacciones son muy complejas, los sistemas son más difíciles de diseñar y construir, y pueden dar problemas de funcionamiento poco eficaz y baja capacidad de mantenimiento

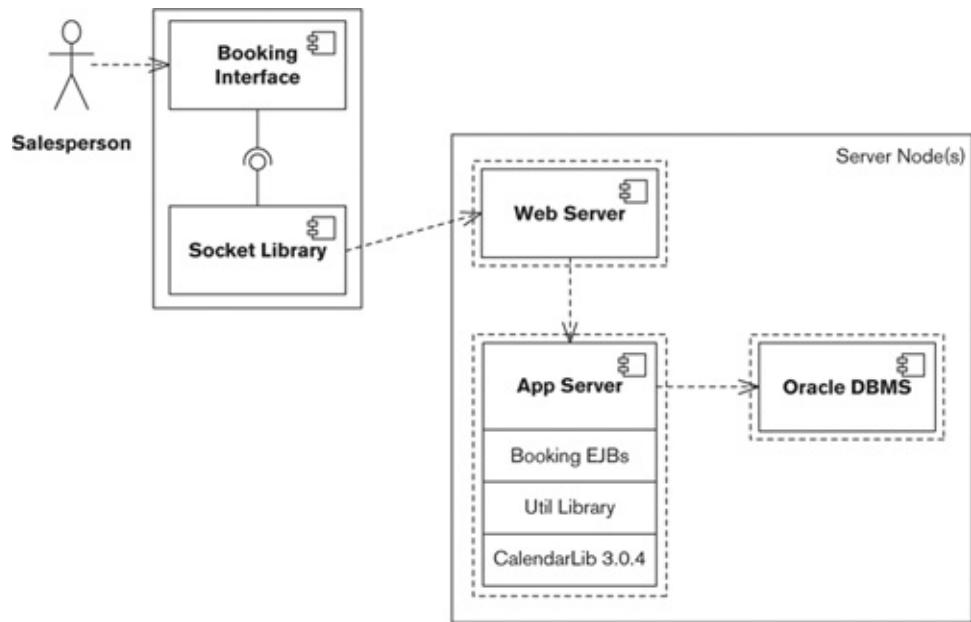


Figura 3.22: Ejemplo de diagrama de una vista funcional sobrecargada. [Fuente:⁴⁸]

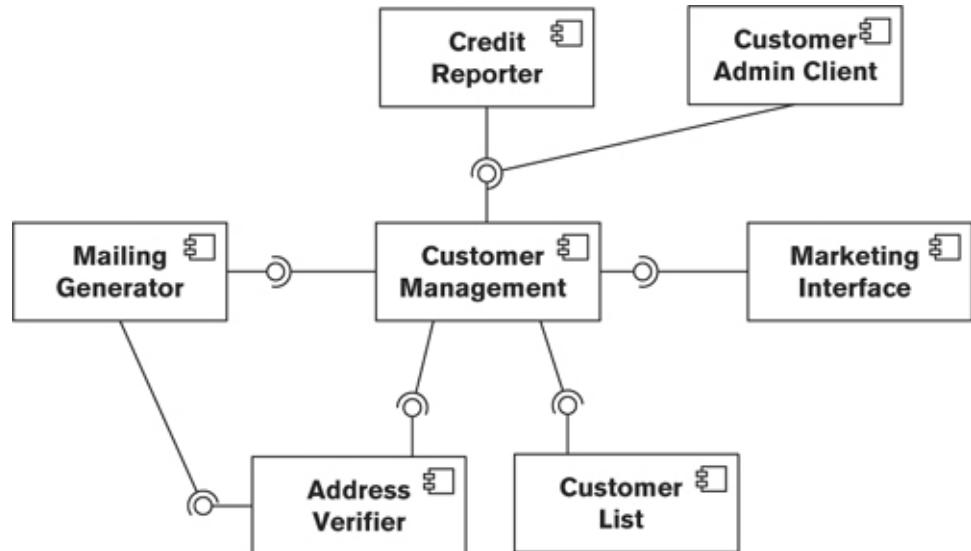


Figura 3.23: Ejemplo de diagrama funcional que adolece del problema del elemento divino. [Fuente:⁴⁹]

Lista de comprobación

Se propone la siguiente:

- ¿Tienes menos de 15 a 20 elementos de nivel superior?

- ¿Tienen todos los elementos un nombre, responsabilidades claras e interfaces claramente definidas?
- ¿Tienen lugar todas las interacciones de elementos a través de interfaces y conectores bien definidos que unan las interfaces?
- ¿Exhiben los elementos un nivel apropiado de cohesión?
- ¿Exhiben los elementos un nivel apropiado de acoplamiento?
- ¿Has identificado los escenarios de uso importantes y los has utilizado para validar la estructura funcional del sistema?
- ¿Has verificado la cobertura funcional de la arquitectura para asegurarte de que cumple con los requisitos funcionales?
- ¿Has definido y documentado un conjunto apropiado de principios de diseño arquitectónico y tu arquitectura cumple con estos principios?
- ¿Has considerado cómo es probable que la arquitectura haga frente a posibles escenarios de cambio en el futuro?
- ¿La presentación de la vista tiene en cuenta las preocupaciones y capacidades de todos los grupos de partes interesadas? ¿Actuará la vista como un vehículo de comunicación eficaz para todos estos grupos?

3.6.8. Adición de “perspectivas” al estándar P1471 basado en puntos de vista

Rozansky hace caer en la cuenta que, a diferencia de los puntos de vista y de sus vistas, que desglosan el sistema en partes en cierta medida independientes, hay una serie de inquietudes que son comunes a muchas de las vistas y deberían reflejarse en la DA como comunes a ellas. Se refieren más bien a requisitos no funcionales, y entre ellos, a criterios de calidad. Aunque algunos los han catalogado como puntos de vista, él destaca que no es oportuno hacerlo así por ser requisitos transversales a distintos puntos de vista.

DEFINICIÓN 3.6.6: Escenarios de calidad del sistema

.- En la descripción de las perspectivas, puede ser necesario el uso de *escenarios de calidad*, los cuales responden al cómo debe reaccionar el sistema a un cambio en el entorno para garantizar uno o más criterios de calidad. No siempre los cambios en el entorno pueden modelarse como estímulos de entrada, como ocurre en los escenarios funcionales.

Para describirlos suele proporcionarse la siguiente información:

1. Descripción general: una breve descripción de lo que el escenario debe ilustrar

2. Estado del sistema: el estado del sistema antes de que ocurra el escenario, si el comportamiento especificado en el escenario depende de ello. Para escenarios de calidad, esto puede necesitar definir aspectos del estado de todo el sistema (como un nivel de carga en todo el sistema) en lugar de la información almacenada en el sistema
3. Entorno del sistema: cualquier observación importante sobre el entorno en el que se ejecuta el sistema, como la falta de disponibilidad de sistemas externos, el comportamiento particular de la infraestructura, las situaciones basadas en el tiempo, etc.
4. Cambios en el entorno: una explicación de lo que ha cambiado en el entorno del sistema que hace que ocurra el escenario. Esto podría ser cambios o fallas en la infraestructura, cambios en el comportamiento del sistema externo, ataques de seguridad, modificaciones requeridas o cualquiera de los otros cambios en el entorno que requieren que el sistema posea una propiedad de calidad particular para tratar con ellos
5. Comportamiento requerido del sistema: una definición de cómo debe comportarse el sistema en respuesta al cambio en su entorno (por ejemplo, cómo debe responder el sistema, desde un punto de vista de rendimiento cuantificable, a un aumento definido en el número de solicitudes que llegan por minuto)

EJEMPLO 3.6.15. Tres escenarios de calidad que sumarizan los datos de entrada**1. Problemas de tamaño en la actualización diaria de datos**

- Descripción general: cómo se comporta el procesamiento del sistema al final del día cuando los volúmenes de datos regulares se superan repentinamente
- Estado del sistema: el sistema tiene estadísticas resumidas en su BD que ya se han procesado, y los elementos de procesamiento del sistema se cargan poco a poco, a la velocidad actual de carga
- Entorno del sistema: el entorno funciona correctamente; los datos llegan a una velocidad constante de 1,000 – 1,500 artículos/hora
- Cambios en el entorno: la tasa de actualización de datos en un día en particular aumenta repentinamente a 4,000 artículos por hora
- Comportamiento requerido del sistema: cuando comienza el procesamiento al final del día, el sistema debe procesar los datos de ese día en un tiempo de procesamiento que no exceda un límite configurable por el sistema. Si se excede, el sistema debería dejar de procesar datos y descartar el trabajo en proceso, dejando el resumen estadístico anterior en su lugar y registrar un mensaje de diagnóstico (incluyendo causa y acción tomada) en el sistema de monitoreo

2. Fallo en la instancia de la base de datos de resúmenes

- Descripción general: cómo se comporta el sistema cuando falla la BD donde quiere escribirse
- Entorno del sistema: el entorno de implementación funciona bien
- Cambios en el entorno: al escribir estadísticas de resumen en la base de datos, el sistema recibe una excepción que indica que la escritura falló (por ejemplo, la BD está llena)
- Comportamiento requerido del sistema: debe dejar de procesar inmediatamente el conjunto de estadísticas en el que está trabajando y cualquier trabajo en progreso. El sistema debe registrar un mensaje fatal en el sistema de monitoreo y apagarse

3. Necesidad de una dimensión adicional de resumen

- Descripción general: cómo el sistema puede hacer frente a la necesidad de ampliar el procesamiento estadístico proporcionado
- Entorno del sistema: el entorno de implementación funciona normalmente, como se entregó inicialmente
- Cambios en el entorno: surge la necesidad de admitir una nueva dimensión en las estadísticas de resumen para resumir las ventas por tipo de opción de pago utilizado
- Comportamiento del sistema requerido: el equipo de desarrollo debe poder agregar la nueva función sin cambiar la estructura general del sistema y con un esfuerzo total de menos de 4 personas-semana

La seguridad: un ejemplo de inquietud “transversal” Para explicarlo pone un ejemplo relacionado con el criterio de seguridad. En su opinión, este criterio de calidad no suele abordarse como debe durante el ciclo de vida del proyecto. Una razón es que no es fácil garantizar un nivel apropiado de seguridad y se tiende a declinar en otros la responsabilidad, cuando es la empresa la que está obligada por ley a garantizar la seguridad en el software que desarrolle.

La razón por la que se trata de una inquietud transversal y por lo tanto no puede considerarse como un punto de vista más es que es una inquietud para distintos puntos de vista:

- Punto de vista funcional.- El sistema de identificar y autenticar a sus usuarios, y defender el sistema frente a ataques externos.
- Punto de vista informacional.- El sistema de controlar distintos tipos de acceso a la información (leer, borrar, actualizar, insertar) y puede requerir aplicar criterios de seguridad con distintos niveles de granularidad.
- Punto de vista operacional.- El sistema debe mantener y distribuir información secreta (palabras clave) según los sistemas de seguridad que en ese momento estén en uso.

Possiblemente también en los puntos de vista de desarrollo, de concurrencia y de despliegue haya algunos aspectos que se vean afectados por la necesidad de garantizar la seguridad.

Perspectivas arquitectónicas Rozansky define el concepto de *perspectiva arquitectónica* de la siguiente forma:

DEFINICIÓN 3.6.7: Perspectiva arquitectónica

Colección de actividades, tácticas y guías arquitectónicas usadas para asegurar que el sistema cumple con un conjunto relacionado de criterios de calidad que deban ser considerados de forma transversal, es decir, por un conjunto diverso de vistas arquitectónicas.

No se trata de nada nuevo, las perspectivas son los requisitos no funcionales, pero lo importante de la propuesta es que se ofrece un mecanismo para que la arquitectura sistematice la implementación necesaria para su cumplimiento.

Para implementar las perspectivas, se añade el concepto de *táctica arquitectónica*, desarrollado por investigadores software del Carnegie Mellon Software Engineering Institute (SEI), y algo modificado por el enfoque de Rozansky, que la define como un enfoque provado que puede usarse para conseguir una propiedad (criterio) de calidad concreto.

Así, para garantizar un criterio de calidad no sólo se debe revisar su cumplimiento en los modelos arquitectónicos sino que se deben seleccionar y probar algunas *tácticas arquitectónicas* que solucionen casos específicos que la arquitectura no pueda abordar.

Se podría hacer una equivalencia entre tácticas y patrones, de forma que las tácticas arquitectónicas son para los requisitos no funcionales como los patrones de diseño son pa-

ra los requisitos funcionales. Pero las tácticas son bastante más generales, son sólo guías generales.

Un ejemplo de táctica arquitectónica para garantizar el rendimiento completo del sistema podría ser definir diferentes prioridades de procesamiento para las distintas partes de la carga de trabajo del sistema y gestionarlas mediante un planificador de procesos basado en prioridades, algo como lo que hacen los sistemas operativos para la gestión de procesos no interactivos de cálculo intensivo.

La perspectiva proporciona un marco para guiar y formalizar el cumplimiento de un criterio de calidad, de forma que desde éste se tenga en cuenta cómo garantizarlo en las distintas vistas arquitectónicas a las que les afecte, tomando las decisiones arquitectónicas que sean necesarias.

Se destacan de ella tres características principales:

- Es un *almacén de conocimiento* de gran utilidad, ayudando a revisar de forma rápida cómo cada modelo arquitectónico garantiza un determinado criterio de calidad sin necesitar usar documentos más detallados
- Es una *guía* eficiente si se trabaja en un dominio de aplicación desconocido y sus inquietudes, problemas y soluciones
- Es un *soporte a la memoria* cuando se conoce el dominio, para garantizar que no se nos olvida nada importante

Las perspectivas deben aplicarse en las etapas más iniciales del diseño de la arquitectura, aunque se haga de manera informal, para que se puedan garantizar mejor los criterios de calidad sin que sea al alto coste de rediseñar por no haberlas tenido en cuenta a tiempo.

Plantilla para definir una perspectiva arquitectónica Se debe ser sistemático a la hora de definir una perspectiva. La propuesta de Rozansky consiste en la consideración de los siguientes aspectos para describir perspectiva en un proyecto concreto (Tabla 3.7).

Perspectivas más importantes También se proponen algunas perspectivas más importantes que deben abordarse en sistemas de información de gran tamaño (Tabla 3.8).

En la realidad, no todas las perspectivas pueden aplicarse a todos los puntos de vista. La Figura 3.24 muestra un ejemplo de aplicación de varias perspectivas a los distintos puntos de vista, mediante una tabla 2x2 donde sólo se colorea una celda si corresponde a una perspectiva y un punto de vista sobre el que se puede aplicar dicha perspectiva.

Como criterio para seleccionar los puntos de vista que abordarán cada perspectiva se aconseja:

⁵⁰Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*, Second Edition (Addison-Wesley Professional, 2011), <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

⁵¹Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*, Second Edition (Addison-Wesley Professional, 2011), <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

Detalle	Descripción
Aplicabilidad	Qué puntos de vista son más probables que se vean afectados. Por ejemplo, si se trata de la capacidad de evolución, el punto de vista funcional estará más afectado que el operacional
Inquietudes	Define los criterios de calidad que son abordados por la perspectiva
Actividades	Define los pasos para aplicar las perspectivas a los distintos puntos de vista y de ahí a las vistas, de forma que se adopten decisiones en el diseño arquitectónico de las vistas para garantizar desde cada vista el criterio de calidad de la perspectiva
Tácticas arquitectónicas	Descripción de las tácticas más importantes para garantizar los criterios de calidad
Problemas y errores	Menciona errores comunes y pautas para reconocerlos y evitarlos
Lista de comprobación	Un listado con todas las cuestiones que no deben olvidársenos para que lo podamos repasar (inquietudes, tácticas, trampas, errores comunes)
Documentación adicional	La descripción de la perspectiva debe ser concisa. Los detalles adicionales pueden describirse en otro lugar y ser referidos aquí

Tabla 3.7: Plantilla para describir la aplicación de una perspectiva en un proyecto concreto.
[Fuente:⁵⁰]

- Tener en cuenta las inquietudes de las partes interesadas en el sistema
- La importancia relativa de los distintos criterios de calidad en el sistema concreto
- La propia experiencia y el juicio profesional

Elaborar esta tabla puede ser útil para el arquitecto software como paso previo a la aplicación de las perspectivas.

Para estas celdas afectadas, deberá darse información (representada con texto en círculos) con el detalle de cómo traducir el criterio de calidad en el contexto del punto de vista concreto (criterios más específicos, inquietudes, guías para el diseño, etc.).

Perspectiva	Descripción
Accesibilidad	Capacidad del sistema de poder ser usado por personas con alguna discapacidad
Ubicuidad(des-localización)	Capacidad del sistema para superar problemas debidos a la localización concreta de sus partes y la distancia entre ellas
Disponibilidad y resiliencia	Asegura que el sistema esté disponible cuando sea necesario y que se restablezca después de posibles fallos
Eficiencia y escalabilidad	Cumpliendo con los requerimientos de rendimiento y respondiendo de forma satisfactoria los incrementos en la carga de trabajo
Evolución	Garantiza que el sistema pueda responder a cambios posibles en el mismo
Usabilidad (facilidad de uso)	Capacidad de facilitar a los usuarios un trabajo eficaz y seguro
Factibilidad (viabilidad de recursos de desarrollo)	Capacidad para que el sistema pueda ser diseñado, construido, lanzado y utilizado dentro de las restricciones de recursos impuestas, tales como personas, presupuesto, tiempo y espacio
Internacionalización	Capacidad del sistema de funcionar de la misma forma independientemente de idiomas, países o grupos culturales
Regulabilidad	Habilidad del sistema para cumplir con las leyes internacionales y regulación quasi-legal (normas éticas), políticas de la industria y otras reglas y estándares
Seguridad	Garantiza el acceso controlado a los recursos del sistema

Tabla 3.8: Perspectivas más importantes en proyectos de gran envergadura [Fuente: [51](#)]

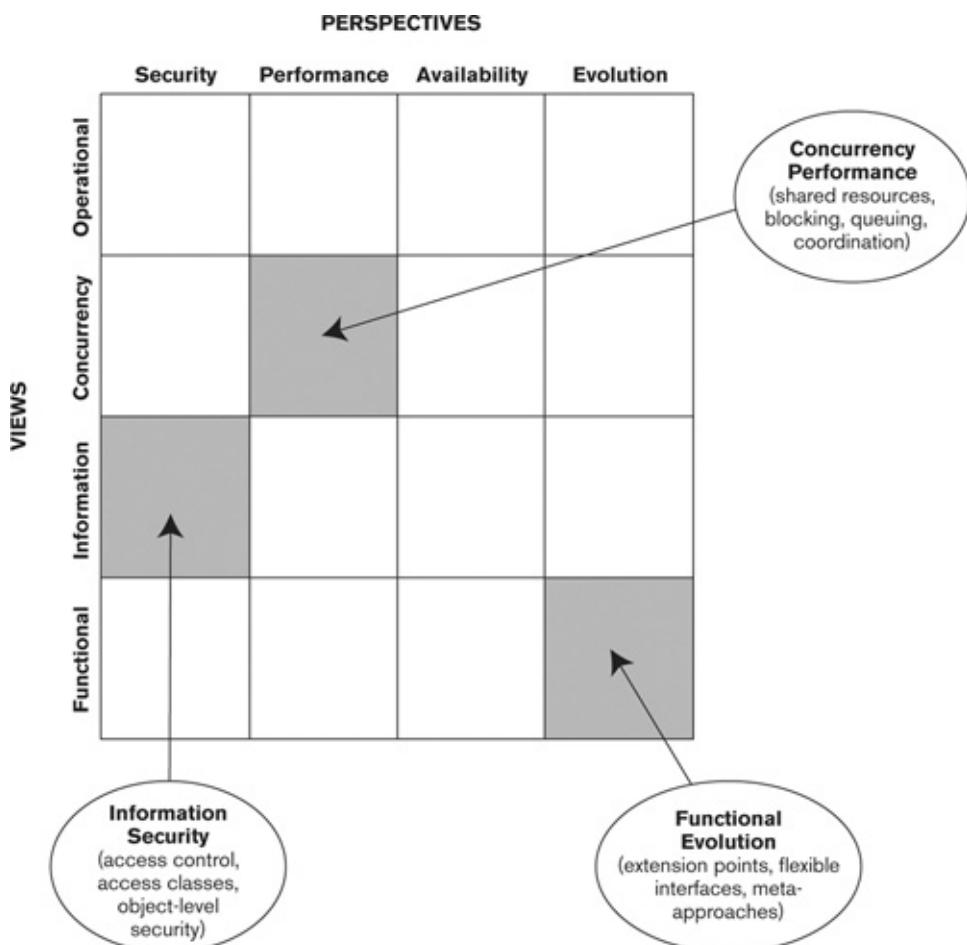


Figura 3.24: Ejemplo de aplicación de perspectivas a puntos de vista. [Fuente: [52](#)]

3.6.9. Descripción detallada de una perspectiva: la perspectiva de seguridad

DEFINICIÓN 3.6.8: Datos sensibles

Se llama *datos sensibles* a aquella información que debe ser protegida frente al acceso no autorizado.

Hoy en día la seguridad es clave en la mayor parte de sistema de información, ya que pueden ser sistemas distribuidos y usar internet u otras redes y necesitan garantizar que sólo puedan acceder a cada recurso los que estén autorizados.

Los especialistas en seguridad utilizan una terminología específica:

- *Principales*.- Los actores o subsistemas software con acceso identificado
- *Recursos*.- Partes del sistema que tienen acceso controlado
- *Políticas*.- Define el acceso legítimo a cada recurso
- *Mecanismos de seguridad*.- Usado por los principales del sistema para obtener el acceso que necesitan

La Figura 3.25 refleja las relaciones entre estos elementos.

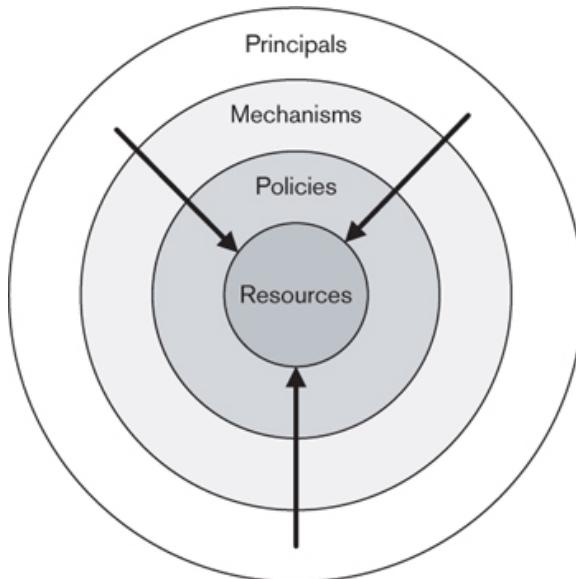


Figura 3.25: Relación entre principales, políticas, mecanismos y recursos. [Fuente:⁵³]

Estos elementos son muy diferentes según el tipo de sistema. En todo caso hay que tener en cuenta que la seguridad no es un estado binario sino un proceso de gestión de riesgos que equilibra los riesgos probables de seguridad con los costes que requiere garantizarlas.

Se describe la perspectiva siguiendo la plantilla propuesta en la Tabla 3.7:

Aplicabilidad Se trata de ver cómo afecta la seguridad a los distintos puntos de vista. La Tabla 3.9 responde teniendo en cuenta los siete puntos de vista considerados hasta ahora.

Punto de vista	Aplicabilidad
Contextual	Permite identificar claramente las conexiones externas al sistema y cómo deben protegerse de un uso malicioso para evitar la vulnerabilidad del sistema, incluso cambiando dichas conexiones
Funcional	Permite identificar los elementos funcionales del sistema que deben protegerse y en consecuencia pueden tener que implementar políticas concretas de seguridad
Informacional	Debe identificar los datos del sistema que deben ser protegidos, y los modelos de datos modificados para implementar los diseños concretos de seguridad, como por ejemplo, dividiéndolos según su sensibilidad
Concurrente	El diseño de seguridad puede señalar la necesidad de aislar distintas piezas del sistema en diferentes elementos de ejecución (como hebras), afectando así a la estructura de la concurrencia del sistema
De desarrollo	Identifica guías y restricciones que los desarrolladores software necesitan conocer para garantizar la política de seguridad.
De despliegue	El diseño de seguridad puede afectar de forma considerable a este punto de vista, incluyendo hardware o software seguro o añadiendo algunas decisiones de despliegue para prevenir los riesgos de seguridad
Operacional	También es muy importante para garantizar las políticas de seguridad la forma en la que se usa el sistema una vez en explotación. Este punto de vista debe dejar muy claras las asunciones y responsabilidades de seguridad de forma que se reflejen en los procesos operacionales

Tabla 3.9: Aplicación de la perspectiva de seguridad a los distintos puntos de vista. [Fuente:⁵⁴]

CRITERIO DE CALIDAD 3.6.1: Confidencialidad

La confidencialidad (considerada aquí como un aspecto dentro de la perspectiva general de seguridad), se define como la capacidad de garantizar el acceso a un recurso sólo a los que están legítimamente autorizados.

⁵⁴Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*, Second Edition (Addison-Wesley Professional, 2011), <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

CRITERIO DE CALIDAD 3.6.2: Integridad

La integridad (considerada aquí como un aspecto dentro de la perspectiva general de seguridad), se define como la capacidad de garantizar que la información no pueda ser modificada de forma no detectable.

CRITERIO DE CALIDAD 3.6.3: Disponibilidad

La disponibilidad (considerada aquí como un aspecto dentro de la perspectiva general de seguridad, más allá de su acepción en el nivel operativo), se define como la capacidad de garantizar que ningún ataque del sistema bloquee el acceso al sistema o a una parte del mismo.

CRITERIO DE CALIDAD 3.6.4: Trazabilidad

La trazabilidad (considerada aquí como un aspecto dentro de la perspectiva general de seguridad), se define como la capacidad de garantizar que se pueda conocer sin ambigüedad la secuencia de pasos que fueron dados para llevar a cabo una acción, retroayendo los pasos hasta llegar al principal que la inició.

Inquietudes Se trata de describir cómo abordar las inquietudes de seguridad, empezando por definir las partes vulnerables (recursos) y terminando por describir los mecanismos concretos de seguridad que serán utilizados. Para un enfoque sistemático, se propone describir cada uno de los siguientes aspectos:

- Recursos.- Las partes que requieren acceso controlado puede requerirlo bien por tener información importante o bien por realizar operaciones sensibles. Se deben establecer los mecanismos de seguridad apropiados (de acceso o de ejecución) para protegerlas.
- Principales.- Es necesario que el sistema permita el acceso identificado de cada principal para poderles dar los *privilegios de acceso* o autorizaciones adecuadas.
- Políticas.- Deben detallar el tipo de acceso de cada tipo de principal (empleados, administradores, gestores, etc.) para cada tipo de información. Además, deben describir cómo se controlará la ejecución de operaciones del sistema sensibles. También definirá restricciones de integridad, tales como reglas y comprobaciones a aplicar en los almacenes de datos y en la protección de documentos frente a cambios no autorizados.
- Amenazas.- Se deben identificar para añadir los mecanismos de seguridad necesarios para afrontarlas, equilibrando asimismo la garantía de la seguridad y la usabilidad del sistema (el índice de riesgo de cada amenaza concreta dependerá del tipo de sistema).
- Confidencialidad.- Si la información no sale del sistema, suele garantizarse mediante el control de acceso. Si hay transmisión a otros sistemas suele garantizarse mediante encriptación.

- Integridad.- Suele garantizarse mediante firma encriptada (firma electrónica o huella digital).
- Disponibilidad.- Para garantizarla hay que diseñar el sistema pensando en los posibles riesgos que pueden derivarse como consecuencia de ataques piratas.

DEFINICIÓN 3.6.9: Contabilización (accounting)

«Las partes implicadas en la transacción (emisor – receptor) tienen que aceptar los datos de la relación creada.» [Minubeinformatica.com](#)

DEFINICIÓN 3.6.10: No repudio

«Informes externos que los sistemas generan sobre eventos, actuaciones, usos, etc de si mismos. Siguiendo y analizando la información generada en estos informes se puede localizar por donde se ha producido un error o ataque e intentar resolverlo. Son los llamados logs del sistema.» [Minubeinformatica.com](#)

- Trazabilidad.- El mecanismo más común para garantizarla en sistemas centralizados es el de contabilización (del inglés, accounting) (Definición 3.6.9). En sistemas distribuidos se suelen usar mensajes encriptados como prueba de que lo envía un principal concreto (no repudio) (Definición 3.6.10).
- Detección y recuperación.- Puede ir más allá de simples acciones tecnológicas, teniendo que involucrar también a personas y establecer procedimientos de acción
- Mecanismos de seguridad.- Para establecerlos se debe contar con expertos en seguridad que conozcan mejor las tecnologías disponibles de seguridad y cómo combinarlas para garantizar la seguridad del sistema

Sin embargo, en este apartado es suficiente con describir los recursos y pasar después a describirlos con más detalle y clasificarlos, continuando con la descripción de las políticas de seguridad, así como las amenazas y los mecanismos de seguridad siguiendo la secuencia de actividades propuesta por Rozansky (ver Figura⁵⁵).

Actividades La Figura 3.26 muestra una propuesta de la secuencia de actividades que deben llevarse a cabo para aplicar la perspectiva de seguridad.

⁵⁵Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*.

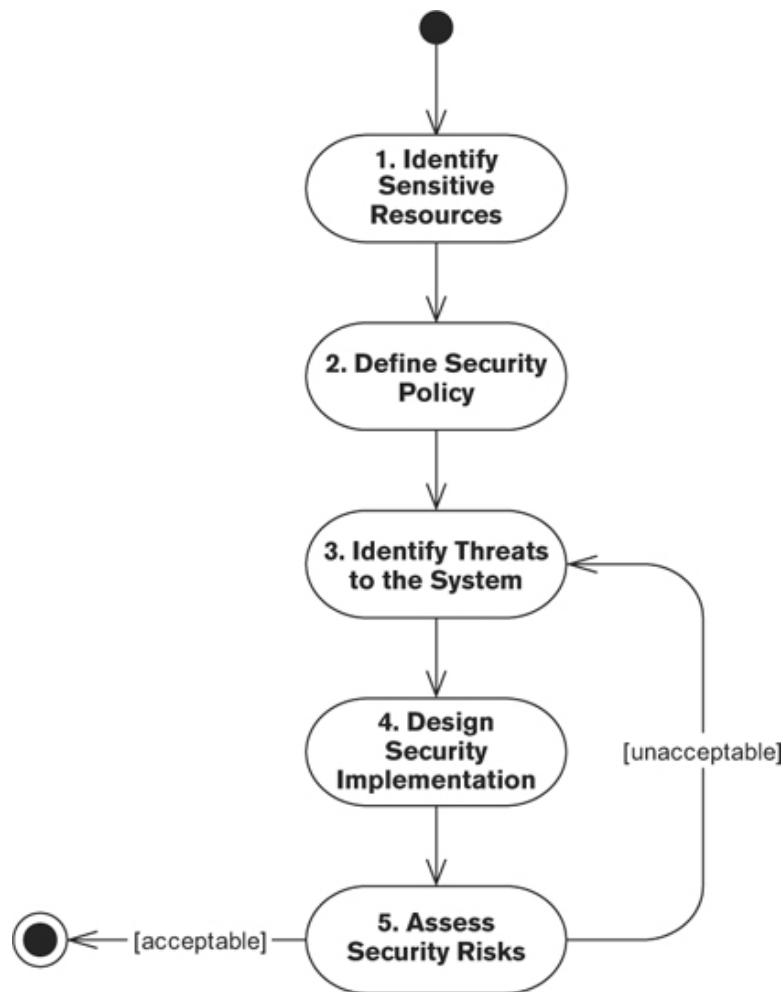


Figura 3.26: Secuencia de actividades a realizar para aplicar la perspectiva de seguridad.
[Fuente:⁵⁶]

Se empieza por una clasificación de los recursos y principales (paso 1) y se sigue con la descripción de las políticas de seguridad a adoptar (paso 2).

Luego se sigue de forma cíclica con la identificación de las amenazas del sistema (paso 3), la descripción de los mecanismos de seguridad para proteger al sistema frente a las amenazas, de forma que se garantice la confidencialidad, integridad y disponibilidad del sistema (paso 4) y la evaluación de los riesgos de seguridad (paso 5), de forma que se volverá al paso 3 si se considera que aún son demasiado altos.

Para cada paso, se definen a su vez un conjunto de sub-actividades, y las notaciones usadas más usuales, tal y como aparece en la Tabla 3.10.

⁵⁷Es una estructura en árbol que categoriza e ilustra las amenazas al sistema y la probabilidad de que ocurran.

⁵⁸Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*, Second Edition (Addison-Wesley Professional, 2011), <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

Actividad principal	Sub-actividades	Notación
1. Identificar recursos sensibles	Clasificar recursos sensibles	Tablas/texto Diagramas usados en p.v. funcional o informativo
2. Definir la política de seguridad	Definir tipos o clases de principales según rol y tipos de accesos. Definir tipos o clases de recursos según uniformidad en el control del acceso. Definir conjuntos de control de acceso (operaciones que pueden realizarse en cada tipo de recurso y tipos de principales que pueden hacerlas). Identificar las operaciones sensibles del sistema y definir los tipos de principales que tienen acceso a ellas. Identificar los requisitos de integridad (situaciones del sistema donde los recursos puedan ser modificados (información) o ejecutados (operaciones) sin permiso.	Tablas
3. Identificar las amenazas al sistema	Identificar las amenazas Caracterizar las amenazas	Tablas/texto <i>Árbol de ataques</i> ⁵⁷
4. Diseñar la implementación de la seguridad	Diseñar una forma de mitigar las amenazas. Diseñar un enfoque de detección y recuperación. Considerar la tecnología. Integrar la tecnología.	Notación de cada vista afectada (Opc.) diagrama UML como en p.v. funcional para mostrar modelo de seguridad global
5. Establecer los riesgos de seguridad	Establecer los riesgos	Tablas

Tabla 3.10: Sub-actividades y notación para cada actividad para abordar la perspectiva de seguridad. [Fuente:⁵⁸]

Paso 1: Identificación de los recursos Para el ejemplo del sistema de comercio electrónico y partiendo del punto de vista funcional, los tipos de recursos que pueden identificarse a partir de los diagramas, en función del tipo de acceso son:

- Gestión de las cuentas de usuarios, y de los clientes en especial
- Gestión del catálogo de productos, y de los precios en especial
- Gestión de las compras

Partiendo del punto de vista de la información, los tipos de recursos que pueden identificarse según su sensibilidad son:

- Registros de cuentas de clientes
- Catálogo de productos
- Precios de los productos
- Registro de operaciones realizadas por clientes (compras)

Los principales pueden también clasificarse (en roles) en función de los recursos a los que pueden acceder:

- Cliente
- Gestor de catálogo
- Vendedor (quien prepara un pedido)
- Repartidor (quien entrega un pedido)
- Administrador
- Superusuario

La Figura 3.27 muestra la Tabla 25-2 Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition* en donde se identifican los recursos en la Tienda Web.

Resource	Sensitivity	Owner	Access Control
Customer account records	Personal information of value for identity theft or invasion of privacy	Customer Care Group	No direct data access
Descriptive product catalog entries	Defines what is for sale and its description; if maliciously changed, could harm the business	Stock Management Group	No direct data access
Pricing product catalog entries	Defines pricing for catalog items; if maliciously or accidentally modified, could harm the business or allow fraud	Pricing Team in Stock Management Group	No direct data access
Business operations on customer account records	Needs to be controlled to protect data access and integrity	Customer Care Group	Access to individual record or all records by authenticated principal
Descriptive catalog operations	Needs to be controlled to protect data access and integrity	Stock Management Group	Access to catalog modification operations by authenticated principal
Pricing catalog modification operations	Needs to be controlled to protect data access and integrity	Pricing Team	Access to price modification operations by authenticated principal, with accountability of changes
***	***	***	***

Figura 3.27: Tabla 25-2 donde se identifican algunos de los recursos sensibles en el sistema de la Tienda Web. [Fuente:⁵⁹]

Paso 2: Definición de la política de seguridad

La Figura 3.28 muestra la Tabla 25-3 Rozanski en donde se identifican las políticas de control de acceso en la Tienda Web.

	User Account Records	Product Catalog Records	Pricing Records	User Account Operations	Product Catalog Operations	Price Change Operations
Data administrator	Full with audit	Full with audit	Full with audit	All with approval and audit	All with audit	All with approval from a product price administrator
Catalog clerk	None	None	None	All	Read-only operations	None
Catalog manager	None	None	None	Read-only operations with audit	All	All with audit
Product price administrator	None	None	None	None	Read-only operations	All with audit
Customer care clerk	None	None	None	All with audit	Read-only operations	None
Registered customer	None	None	None	All on own record	Read-only operations	None
Unknown Web-site user	None	None	None	None	Read-only operations	None

Figura 3.28: Tabla 25-3 donde se muestra parte del resultado de definir la política de seguridad para el sistema de Tienda Web. [Fuente:⁶⁰]

En este paso se han descrito 7 tipos distintos de principales según el control de acceso y 5 tipos distintos de recursos sensibles.

Paso 3: Identificación de las amenazas del sistema

En este paso hay que construir un *modelo de amenazas*, que parte de cada recurso identificado y considera cada posible amenaza, el impacto (consecuencias) en el sistema y la probabilidad de que ocurra.

Para elaborarlo, debe responderse a las siguientes preguntas:

- ¿Quién es probable que quiera saltarse la política de seguridad?
- ¿Qué motivación tiene el atacante para atacar el sistema?
- ¿Cómo tratará de saltarse la política de seguridad?
- ¿Cuáles son las principales características del atacante (sofisticación, compromiso, recursos, etc.)?
- ¿Cuáles son las consecuencias de que se salte la política de esta forma?

Hay que considerar ataques tanto externos como internos, cómo afectan los ataques al entorno (por ejemplo al proveedor de servidores –hosting–, o a un entorno de computación en la nube). También se debe contar con los recursos de seguridad implementados por los proveedores de servicios o por el entorno en general.

La notación más común es la de texto y tablas pero también pueden usarse árboles de ataques: resultados de amenazas por categorías y probabilidad de que ocurran, puestos en forma jerárquica.

Se pueden considerar dos subactividades para este paso:

- Identificar las amenazas, considerando la sensibilidad de los recursos y los posibles tipos de atacantes

- Caracterizar las amenazas, identificando los recursos objetivo, las consecuencias y la probabilidad del ataque

El cuadro 3.6.16 muestra el ejemplo de un árbol de ataque para la amenaza de obtener los detalles de la tarjeta de crédito de un cliente en un sistema de comercio electrónico. En esta notación, suele empezarse con un paso previo: identificar los objetivos de cada atacante. Se definirá un árbol por cada objetivo. Un objetivo incluye el conjunto de amenazas al sistema que el atacante puede intentar para lograrlo.

El árbol de ataque es un método de representación del modelo de amenaza de un sistema. Se basa en la técnica de los árboles de fallos en el diseño de sistemas críticos.

Un árbol de ataque representa los posibles ataques que el sistema puede sufrir para que un atacante logre un objetivo particular. La raíz del árbol es el objetivo que el atacante está tratando de lograr, y las ramas del árbol clasifican los diferentes tipos de ataques que el intruso podría intentar para lograr el objetivo.

Pueden representarse gráficamente (como una estructura de árbol con nodos y enlaces) o textualmente usando títulos numerados anidados.

EJEMPLO 3.6.16. Árbol de ataque

Este es un posible árbol de ataque con el objetivo de extraer los datos de la tarjeta de crédito del cliente de un sitio web de comercio electrónico:

Objetivo: obtener los datos de la tarjeta de crédito del cliente.

1. Extraer detalles de la base de datos del sistema

- a) Acceder a la base de datos directamente
 - 1) Descifrar / adivinar contraseñas de bases de datos
 - 2) Romper / adivinar las contraseñas del sistema operativo que permiten evitar la seguridad de la base de datos
 - 3) Explotar una vulnerabilidad conocida en el software de la base de datos
- b) Acceder a los detalles a través de un miembro del personal de administración de la base de datos
 - 1) Sobornar a un administrador de base de datos (DBA)
 - 2) Realizar ingeniería social por teléfono / correo electrónico para engañar al DBA para que revele detalles

2. Extraer detalles de la interfaz web

- a) Configurar un sitio web ficticio y enviar por correo electrónico a los usuarios la URL para engañarlos para que introduzcan los detalles de la tarjeta de crédito
- b) Descifrar / adivinar contraseñas para cuentas de usuario y extraer detalles de la interfaz web del usuario
- c) Enviar a los usuarios un programa troyano por correo electrónico para grabar pulsaciones de teclas / interceptar el tráfico web
- d) Atacar el servidor de nombres de dominio para secuestrar el nombre de dominio y usar el ataque de sitio ficticio de 2.1
- e) Atacar el software del servidor del sitio directamente para tratar de encontrar lagunas en su seguridad o configuración o para aprovechar una vulnerabilidad conocida en el software

3. Encontrar detalles fuera del sistema

- a) Realizar ingeniería social por teléfono / correo electrónico para que el personal de servicio al cliente revele los detalles de la tarjeta
- b) Dirigir un ataque de ingeniería social a los usuarios mediante el uso de detalles públicos del sitio para hacer contacto (ver también 2.1)

Paso 4: Diseño de los mecanismos de seguridad a implementar

Hay que considerar tecnologías específicas para garantizar la política de seguridad defendiendo al sistema de los ataques identificados. Algunos ejemplos son: cortafuegos, comunicación SSL, criptografía, etc.

Este proceso de diseño da como resultado una serie de decisiones que deben incorporarse en la arquitectura, afectando probablemente a los puntos de vista funcional, de información, de implementación y operacional.

El cuadro 3.6.17 muestra un ejemplo de medidas de seguridad a adoptar para las amenazas identificadas en el árbol de ataque del ejemplo 3.6.16, en un sistema de comercio electrónico.

Hay que notar que este paso no suele incluir una descripción de tecnologías a usar sino más bien en explicar cómo se garantiza la seguridad.

EJEMPLO 3.6.17. Diseño de medidas de seguridad a adoptar

- Aislar las máquinas de la base de datos de la red pública utilizando la tecnología de firewall de red
- Aislar las partes sensibles a la seguridad del sistema de la red pública utilizando la tecnología de firewall de red
- Analizar las rutas en el sistema para verificar si hay vulnerabilidades posibles
- Organizar pruebas de penetración para ver si los expertos pueden encontrar formas de entrar en el sistema
- Identificar una estrategia de detección de intrusos que permita reconocer las violaciones de seguridad
- Capacitar al personal de administración y servicio al cliente (de hecho, probablemente todo el personal) para evitar ataques de ingeniería social y acatar estrictos procedimientos de protección de la privacidad de la información del cliente
- Diseñar el sitio Web para que una cantidad mínima de información del usuario (idealmente, ninguna) sea visible públicamente
- Diseñar el sitio Web para que la información confidencial (por ejemplo, números de tarjetas de crédito) nunca se muestren en su totalidad (por ejemplo, muestre solo los últimos cuatro dígitos para permitir que los usuarios legítimos identifiquen sus tarjetas en las listas)
- Aplicar de forma rutinaria actualizaciones software relacionadas con la seguridad a todo el software de terceros utilizado en el sistema
- Revisar el código del sistema para detectar vulnerabilidades de seguridad utilizando herramientas de análisis e inspección experta
- Recordar constantemente a los usuarios las precauciones de seguridad que deben tomar (por ejemplo, no revelar las contraseñas a nadie, incluido su personal; verificar las URL antes de ingresar información, etc.)

Paso 5: Evaluación de los riesgos de seguridad

No hay que olvidar que la seguridad nunca puede garantizarse de forma total. En este paso hay que hacer balance de riesgo/coste según las medidas adoptadas. Si el resultado es demasiado costoso o los riesgos no tratados demasiado altos, debe volverse al paso 3 de identificación de las amenazas del sistema.

La Figura 3.29 muestra la Tabla 25-4 Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition* en donde se muestra

como ejemplo la evaluación de tres de los riesgos identificados en el sistema de la Tienda Web.

Risk	Estimated Cost	Estimated Likelihood	Notional Cost
Attacker gains direct database access	\$8,000,000	0.2%	\$16,000
Web-site flaw allows free orders to be placed and fulfilled	\$800,000	4.0%	\$32,000
Social-engineering attack on a customer service representative results in hijacking of customer accounts	\$4,000,000	1.5%	\$60,000
***	***	***	***

Figura 3.29: Tabla 25-4 donde se muestra la evaluación de tres de los riesgos identificados en el sistema de la Tienda Web. [Fuente:⁶¹]

Tácticas arquitectónicas

Se proporciona la siguiente lista de posibles tácticas:

1. Aplicar los principios de seguridad que gocen de reconocimiento. Algunos ejemplos son:
 - Asignar el mínimo nivel de privilegios posible
 - Asegurar los accesos más débiles
 - Defensa en profundidad (esquema de protección por capas)
 - Separar y modularizar por responsabilidades
 - Hacer diseños de seguridad simples, que faciliten su análisis
 - No apoyarse en la oscuridad, sino asumir que los posibles atacantes puedan conocer el sistema tan bien como nosotros
 - Usar por defecto los criterios de seguridad (para claves, permisos de acceso, etc.)
 - Seguridad en fallos, no sólo cuando el sistema funciona bien
 - Asumir que las entidades externas no son confiables
 - Auditarse/monitorizar eventos sensibles
2. Autenticar a los principales (personas, ordenadores, subsistemas, etc.)
3. Autorizar el acceso
4. Asegurar la privacidad de la información
5. Asegurar la integridad de la información
6. Asegurar la trazabilidad

7. Proteger la disponibilidad
8. Integrar las tecnologías de seguridad
9. Proporcionar administración de la seguridad (forma parte del punto de vista operacional)
10. Usar la infraestructura de seguridad de terceras partes

Problemas y errores comunes

Se destacan los siguientes:

- Políticas de seguridad complejas
- Tecnologías de seguridad disponibles no probadas
- Sistema no diseñado para responder frente a fallos
- Ausencia de facilidades de administración de seguridad
- Enfoque dirigido por la tecnología
- Fallo al considerar las fuentes para medir tiempos
- Exceso de confianza en la tecnología (nunca estamos seguros 100 %)
- Requisitos o modelos de seguridad no bien definidos
- Considerar la seguridad para el final, sin tenerla en cuenta desde el inicio de la DA
- Ignorar las amenazas internas
- Asumir que el cliente es seguro
- Embeber la seguridad en el código de la aplicación, de forma que el modelo de seguridad queda implementado de forma diseminada en distintas partes del código de la aplicación
- Seguridad por piezas, en vez de considerarla y abordarla como un todo
- Uso de tecnologías de seguridad ad hoc

Lista de comprobación

Se proporcionan dos listas, una para capturar todos los requerimientos de seguridad:

- ¿Has identificado los recursos sensibles del sistema?
- ¿Has identificado los conjuntos de principales que necesitan acceder a los recursos?
- ¿Has identificado las necesidades que tiene el sistema de garantizar la integridad de la información?
- ¿Has identificado las necesidades de disponibilidad del sistema?
- ¿Has establecido una política de seguridad para definir las necesidades de seguridad del sistema, junto con los principales y los permisos de acceso que tiene cada uno a cada recurso, y cuándo debe comprobarse la integridad de la información?
- ¿Es la política de seguridad lo más simple posible?
- ¿Has recorrido un modelo formal de amenazas para identificar los riesgos de seguridad del sistema?
- ¿Has considerado tanto las amenazas externas como las internas al sistema?
- ¿Has considerado cómo el entorno de despliegue del sistema alterará las amenazas del sistema?
- ¿Has recorrido escenarios de ejemplos junto con las partes interesadas en el sistema de forma que puedan entender la política de seguridad planificada y los riesgos del sistema?
- ¿Has revisado los requisitos de seguridad con expertos externos en seguridad?

La otra lista proporcionada es directamente aplicable en la DA:

- ¿Has abordado cada amenaza del modelo de amenazas con la profundidad necesaria?
- ¿Has usado lo más posible las tecnologías de seguridad de terceras partes?
- ¿Has realizado un diseño global integrado de la solución dada para garantizar la seguridad?
- ¿Has considerado los principios estándares de seguridad a la hora de diseñar la infraestructura de seguridad?
- ¿Es tu infraestructura de seguridad lo más simple posible?
- ¿Has definido una forma de identificar brechas de seguridad y de recuperar el sistema frente a ellas?

- ¿Has aplicado los resultados de la perspectiva de seguridad a todos los puntos de vista afectados?
- ¿Han revisado tu diseño de seguridad expertos externos en seguridad?

3.6.10. Descripción detallada de otra perspectiva: la perspectiva de evolución

El software, como su nombre indica (“soft”), es “flexible”, de forma que las partes interesadas esperan que un sistema software pueda evolucionar muy rápidamente. Por otro lado, a menudo hay que hacer cambios como consecuencia de requisitos mal comprendidos que el usuario constata cuando empieza a usar el sistema, o el cambio comercial rápido. El enfoque iterativo permite a los usuarios comenzar a usar algunas partes de él mucho antes de estar terminado, proporcionando retroalimentación temprana a los desarrolladores. El problema es la presión constante mientras no se termina por completo, de cambiar el comportamiento del sistema, con la consiguiente necesidad en algunos casos de cambiar su arquitectura.

El software es fácil de cambiar sólo si el cambio se considera explícitamente durante su desarrollo.

DEFINICIÓN 3.6.11: Evolución

Conjunto de todos los posibles tipos de cambios que un sistema puede experimentar durante su vida útil.

La perspectiva Evolución aborda las preocupaciones relacionadas con el manejo de la evolución durante la vida útil de un sistema y, por lo tanto, es relevante para la mayoría de los sistemas de información a gran escala debido a la cantidad de cambios que la mayoría de los sistemas necesitan manejar.

CRITERIO DE CALIDAD 3.6.5: Capacidad de evolución (sistema evolutivo)

La capacidad de evolución de un sistema es su flexibilidad ante cambios inevitables en fase de explotación, con un aumento moderado de los costes de desarrollo necesarios para proporcionar tal flexibilidad

La perspectiva pretende que se cumpla con el criterio de calidad *capacidad de evolución*. Tal y como hicimos con la perspectiva de seguridad, se describirá esta perspectiva siguiendo la plantilla propuesta en la Tabla 3.7.

Aplicabilidad

Se trata de ver cómo afecta la evolución a los distintos puntos de vista, considerando que afectará más a sistemas de mayor uso, tanto intensivo como extensivo. La Tabla 3.11 responde teniendo en cuenta los siete puntos de vista considerados hasta ahora.

Punto de vista	Aplicabilidad
Contextual	Puede necesitar mostrar entidades externas, interfaces o interacciones que formarán parte sólo en versiones futuras del sistema
Funcional	Debe reflejarse la evolución a nivel funcional si los requisitos de evolución son significativos
Informacional	Debe hacerse un modelo informacional flexible si se requiere evolución de la información o del entorno
Concurrente	Las necesidades evolutivas pueden condicionar el empaquetamiento de algún elemento en particular o alguna restricción en la estructura concurrente (v.g., que sea muy simple)
De desarrollo	Los requisitos evolutivos pueden tener alto impacto sobre el entorno que desarrollo que se necesita definir (v.g. forzando guías de portabilidad)
De despliegue	Generalmente la evolución no afecta a esta vista
Operacional	Generalmente la evolución tiene poco impacto en esta vista

Tabla 3.11: Aplicación de la perspectiva de evolución a los distintos puntos de vista. [Fuente:⁶²]

Inquietudes

Se destacan las siguientes:

- Gestión de productos.- En las metodologías ágiles de desarrollo de software (Agile, Scrum, Extreme Programming (XP), etc.) se ha incorporado la idea de ver el software como producto futuro en el mercado, estudiando las necesidades del futuro cliente, así como las amenazas y las oportunidades del entorno donde se utilizará para realizar una hoja de ruta para planificar y supervisar su desarrollo. El papel de propietario del producto, a menudo lo desempeña el arquitecto, que debe trazar el rumbo futuro para el sistema que está desarrollando.

Ya sea que se reconozca formalmente, como en las metodologías ágiles, o no, la gestión del producto es importante porque proporciona un contexto y una dirección para todos los cambios que ocurren en el sistema. Esto ayuda a que los cambios potenciales sean priorizados sistemáticamente y les permite ser considerados en el contexto de una hoja de ruta, para evitar el desarrollo de un conjunto de características incoherentes.

- Magnitud del cambio.- Cuando se realiza la DA de un sistema pensando que sobre él sólo se realizarán pequeños cambios, puede tener que enfrentarse a grandes costos si se tuviera que enfrentar en el futuro a cambios mucho mayores que incluso pueden llevar a la realización de un sistema completamente nuevo.

⁶²Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition* (Addison-Wesley Professional, 2011), Cap. 28, <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

- Dimensiones del cambio.- Es necesario identificar las dimensiones de cambio requeridas, para poder acotar mejor la evolución del sistema. Entre ellas se proponen las siguientes:
 - Evolución funcional: incluye cualquier cambio en las funciones que proporciona el sistema, desde simples correcciones de defectos en un extremo de la escala hasta la adición o reemplazo de subsistemas completos en el otro
 - Evolución de la plataforma: muchos sistemas necesitan evolucionar en términos de las plataformas de software y hardware en las que se implementan. Esto puede incluir migración de plataformas (v.g. de servidores basados en Windows a servidores basados en Linux), así como la ampliación de las plataformas que el sistema puede usar (v.g., transferencia de productos a nuevas plataformas, ampliación de las plataformas cliente existentes basadas en PC a otras basadas en la Web o uso de apps para móviles)
 - Evolución de la integración: la mayoría de los sistemas están integrados en otros, de forma que los cambios de estos otros obligan a que evolucione el nuestro. No tiene que cambiar la funcionalidad pero sí la forma en como se integra con otros sistemas
 - Crecimiento en el uso: la mayoría de los sistemas exitosos experimentan un crecimiento en el uso durante su vida útil que puede deberse a muchos factores, como un aumento en el número o la complejidad de las transacciones, un aumento en el número de usuarios o la necesidad de administrar y almacenar grandes cantidades de datos. Si el sistema proporciona un servicio de Internet exitoso, este crecimiento podría ser sustancial e impredecible
- Probabilidad del cambio.- Identificar los distintos tipos de cambios que podrían ser necesarios es algo fácil, pero evaluar la probabilidad de que los cambios sean realmente necesarios puede ser mucho más difícil. Puesto que ser flexible a los cambios agrega complejidad y gastos, es importante poder afinar estas probabilidades
- Temporización del cambio.- Estimar el momento probable para realizar el cambio requerido también es una preocupación importante. Cuanto más lejos esté la necesidad de un cambio, menos probable es que el cambio sea realmente necesario en su forma actualmente identificada. Los requisitos para los cambios que no tienen fecha de entrega asociada pueden ser de menor prioridad que los cambios con fechas firmes a corto plazo adjuntas
- Cuándo pagar por el cambio.- Hay dos estrategias para planificar el cambio en nuestro sistema:
 1. Diseñar el sistema más flexible posible ahora para facilitar el cambio posterior. Esto se identifica mejor con el enfoque de metasistema, donde la estructura y las funciones de información del sistema se definen en tiempo de ejecución mediante datos de configuración

2. Crear el sistema más simple posible para satisfacer las necesidades inmediatas y enfrentar el desafío de hacer cambios solo cuando sea absolutamente necesario. Esta es más la idea de las metodologías ágiles (por ejemplo los mantras de XP de “Haz el sistema más simple posible” y “No lo vas a necesitar”, que captan la lección de que tratar de adivinar el futuro y construir el sistema más flexible posible es un negocio costoso, arriesgado y complejo)

Una de las principales diferencias entre estas dos estrategias es cuándo se paga por el cambio. El desarrollo de sistemas altamente flexibles cuesta mucho más que el desarrollo de sistemas simples y rígidos, por lo que el costo del cambio se carga al principio del ciclo de vida del sistema si sigue la primera estrategia. La compensación es que espera que estos costos iniciales se paguen con cambios más baratos y rápidos más adelante. Desarrollar el sistema más simple posible cuesta menos por adelantado porque es más simple y rápido de entregar, pero cada cambio posterior probablemente costará más porque no tiene un mecanismo existente para implementarlo.

Conseguir el equilibrio correcto entre estas dos posiciones extremas evita el desperdicio del esfuerzo inicial o los enormes costos de cambio posteriores y nos ayuda a encontrar una posición que minimice los costos generales de desarrollo.

- Cambios dirigidos por factores externos.- No todos los cambios están bajo nuestro control o los de las partes interesadas más inmediatas, algunos pueden ser impuestos por personas o grupos fuera de nuestra esfera de influencia, como por ejemplo un nuevo jefe del departamento TIC que cambie a una estrategia de “comprar en vez de construir”.

Los ejemplos de cambio impulsado externamente incluyen los siguientes:

- El final de la vida útil de los componentes hardware o software que se planea usar como parte de la arquitectura. Si la empresa exige que los sistemas solo puedan ejecutarse en hardware y software admitidos por el proveedor, éstos deben poder proporcionarnos la hojas de ruta para sus productos que identifiquen cuándo es probable que termine su vida útil
 - Cambios en las interfaces con entidades externas, como pasar a un nuevo protocolo, formato de datos, contenido de datos o modelo de interacción
 - Cambios en la regulación externa, que pueden conducir a requisitos más estrictos para la continuidad del negocio, validación, retención de datos, auditoría o control
 - Cambio organizacional que puede conducir a diferentes prioridades, requisitos modificados o cambios en la población de usuarios y el perfil de transacciones
-
- Complejidad del desarrollo.- En casi todos los casos, el apoyo a la evolución aumenta la complejidad del diseño de un sistema, a veces en gran medida y también puede traer problemas relacionados con la fiabilidad del sistema y el tiempo requerido para entregar las primeras partes del sistema. En algunos casos, la complejidad puede incluso convertirse en un obstáculo para la evolución del sistema

- Preservación del conocimiento.- Una preocupación importante para cualquier sistema es cómo preservar el conocimiento requerido para realizar cambios significativos en el sistema a medida que pasa el tiempo, pues las personas se trasladan a otros proyectos, los recuerdos se desvanecen y los entornos técnicos disponibles cambian
- Fiabilidad del cambio.- Desde la corrección de errores más simple hasta la remodelación más compleja, cualquier cambio en el sistema puede tener un impacto negativo en el sistema implementado, por lo que es esencial contar con un conjunto de procesos y tecnologías para hacer que este proceso sea lo más fiable posible. Las pruebas automatizadas, los procesos repetibles y bien entendidos, los entornos de desarrollo estables y la gestión efectiva de la configuración son factores clave para abordar esta preocupación a medida que el sistema evoluciona

Actividades

Se proponen 4 actividades en esta perspectiva, tal y como se muestra en el diagrama de flujo de la Figura 3.30.

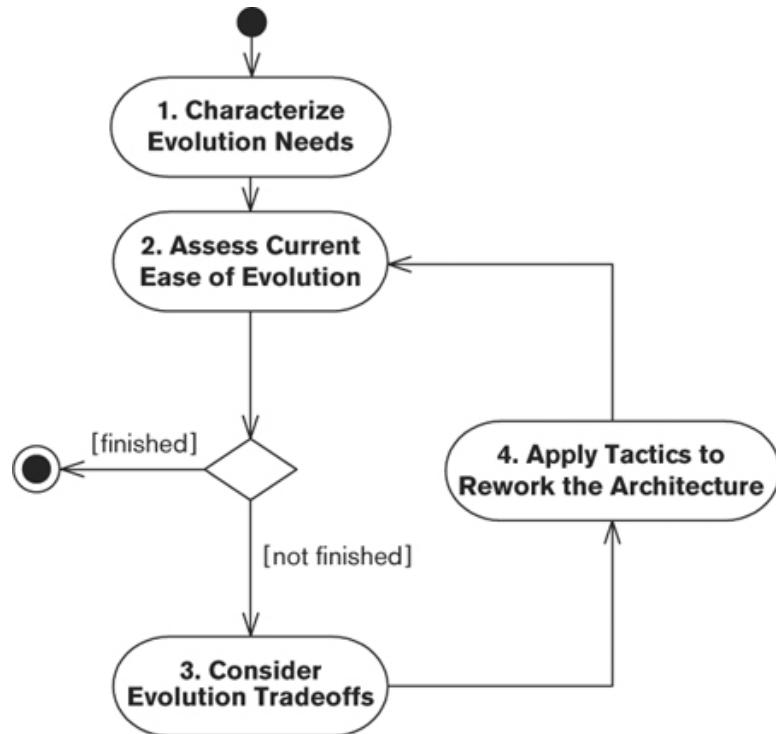


Figura 3.30: Curso de actividades a realizar para describir un sistema desde la perspectiva de evolución. [Fuente:⁶³]

Paso 1: Caracterizar las necesidades evolutivas

En este paso debemos volver a la especificación de requisitos y determinar qué es probable que tenga que cambiar con el tiempo. A menudo en la especificación de requisitos no aparece de forma explícita nada relacionado con la evolución del sistema. Debemos buscar en los requisitos algunos indicios de los siguientes tipos:

- Funciones diferidas: cualquier parte de los requisitos del sistema que defina explícitamente extensiones futuras, o funciones que no necesitan ser entregadas inicialmente
- Lagunas en los requisitos: probablemente requisitos de evolución disfrazados, que no pudieron definirse inicialmente debido a un análisis de requisitos incompleto
- Requisitos vagos o indefinidos: que indican que esta área del sistema no se comprende bien
- Requisitos abiertos: por ejemplo, términos tales como “similar a” o “incluyendo” o “etc.”, en la definición de requisitos del sistema, sugieren que se requerirán extensiones similares a los casos especificados explícitamente

Para cada requisito evolutivo identificado, debemos describirlo utilizando la siguiente plantilla:

1. Tipo de cambio requerido: Se debe clasificar cada tipo de evolución en una de las dimensiones de cambio descritas anteriormente (funcional, de plataforma, de integración o de crecimiento)
2. Magnitud de cambio requerida: Ahora establecemos cuánto esfuerzo necesitará cada tipo de evolución. ¿Es sólo corrección de defectos, o se requerirán cambios a gran escala y de alto riesgo en el sistema? Una forma útil de presentar esto es el esfuerzo requerido en proporción al esfuerzo inicial de desarrollo del sistema
3. Probabilidad de cambio: Se trata de evaluar la probabilidad de que cada uno de los tipos de cambio identificados sea realmente necesario. Esto permite concentrarnos en aquéllos que tienen más probabilidades de ocurrir
4. Escala de tiempo de los cambios requeridos: ¿Se requieren los cambios en un calendario concreto e inmediato (una entrega por fases)? ¿O son necesidades vagas de posibles cambios futuros a hacer dependiendo de factores externos (como el crecimiento del sistema)?

A partir de esta descripción debemos priorizarlos según la importancia global y el tipo de evolución que las partes interesadas esperan del sistema. Una propuesta es ordenarlos dividiendo su magnitud relativa entre el número de meses que pensamos que quedan para que se necesite que el requisito esté implementado y enfocar el esfuerzo en los dos primeros.

Notación: Basta un enfoque de *texto y tablas*.

Paso 2: Evaluar la facilidad para evolucionar en la actualidad

El objetivo de esta actividad es llegar a saber si los cambios requeridos pueden realizarse en el tiempo previsto a un coste razonable. Para ello debemos revisar los requisitos de evolución identificados (especialmente los de mayor prioridad) y pensar (sin identificar los detalles) cómo debería cambiar el sistema para cumplir con el requisito (magnitud, dificultad y riesgo del conjunto de cambios que habría que hacer).

Notación: Descripción textual.

Paso 3: Considerar las contrapartidas de la evolución

Debe considerarse si se debe hacer el esfuerzo de crear un sistema flexible durante el desarrollo inicial o si diferir este esfuerzo hasta que se requieran cambios en el sistema. La decisión depende en gran medida del tipo de sistema, la probabilidad de que los cambios sean realmente necesarios y el nivel de confianza que tiene en poder hacer cambios importantes de forma fácil cuando sea necesario, en lugar de durante el desarrollo inicial.

El resultado de este paso es describir la decisión tomada (cómo evolucionará el sistema y en qué punto se colocará el soporte para la evolución en el sistema).

Notación: Descripción textual.

Paso 4: Revisar la arquitectura

Considerando la mejor estrategia de evolución identificada, se deben cambiar las vistas (puntos de vista) afectadas.

Tácticas arquitectónicas

Aislamiento de cambios

Los cambios pequeños, por ejemplo los que afectan a un sólo módulo software, no suelen ser problemáticos. El problema aparece cuando sus efectos se propagan a través de varias partes diferentes del sistema simultáneamente.

El desafío arquitectónico consiste en diseñar una estructura de sistema sólida para que los cambios requeridos estén tan confinados como sea posible. Los siguientes principios generales de diseño pueden ayudarnos a localizar los efectos del cambio:

- **Encapsulación:** elementos fuertemente encapsulados con interfaces bien definidas y flexibles ayudan a aislar el cambio. Si las estructuras de datos internas de cada elemento no son visibles a sus clientes, los cambios internos a un elemento no se propagarán hacia fuera

- Separación de inquietudes (bajo acoplamiento): es más fácil aislar los cambios cuando las tareas funcionales se asignan a elementos concretos en vez de disgregarlas entre varios de ellos
- Cohesión funcional. también es más fácil aislar un cambio sin existe una alta cohesión, es decir, todas las funciones de un elemento están fuertemente relacionadas entre sí
- Mínima redundancia (punto único de definición): tanto datos como código y configuraciones deben definirse/implementarse una sola vez, para evitar tener que hacer cambios en varias partes diferentes del sistema, que, si no se hace bien, pueden llevar a inconsistencias

Crear interfaces extensibles

Los cambios en las interfaces son los que tienen mayor propagación y por tanto los más costosos. Por ejemplo, agregar un parámetro obligatorio a una función de uso frecuente implica cambiar cada parte de código que llame a esa función, recodificar y volver a probarla. Por tanto, vale la pena invertir en el diseño de cierto nivel de flexibilidad en las interfaces si parece probable que el sistema experimente cambios significativos.

Algunas técnicas que podemos usar incluyen las siguientes:

- Sustituir las APIs que tienen un gran número de parámetros individuales por otras que pasan objetos u otros tipos de datos estructurados. Por ejemplo, un método *CrearEmpleado* con argumentos: nombre y apellido del empleado, fecha de nacimiento y DNI podría reemplazarse por un método que tenga como único argumento un objeto *Empleado*
- Podemos utilizar un enfoque similar con interfaces de información. Por ejemplo, usando una tecnología de mensaje autodescriptivo como XML para definir formatos de mensaje, y permitiendo añadir elementos opcionales al mensaje, de forma que se puedan ampliar los mensajes con poco o ningún impacto en los elementos del sistema que no necesitan usar la forma extendida de la interfaz

Debemos tener en cuenta que estos enfoques no están exentos de costes. Llevar la flexibilidad de la interfaz al extremo implicaría eliminar por completo la escritura estática de sus interfaces y establecer los tipos de todos los parámetros de solicitud en tiempo de ejecución. Un enfoque tan flexible puede ser más difícil de entender y probar y también puede ser menos eficiente. Es algo parecido a lo que hacen los lenguajes no tipados. También puede introducir muchos problemas sutiles en el sistema porque es difícil verificar la información que falta en un momento dado.

Los detalles para crear interfaces flexibles dependen del entorno tecnológico y del dominio del problema del sistema. Sin embargo, es importante para las inquietudes de evolución, considerar el grado de flexibilidad que se requiere en las interfaces más importantes y cómo lograrlo.

Aplicar técnicas de diseño que faciliten el cambio

Se dan aquí una serie de principios, estilos y patrones de diseño que pueden ayudar a que un sistema sea más fácil de cambiar:

- Los patrones de abstracción y estratificación facilitan el cambio de una parte del sistema con un impacto mínimo en otras
- Los patrones de generalización facilitan el manejo de nuevos casos de uso o tipos de datos, al especializar la funcionalidad de propósito general existente de una manera apropiada para el nuevo caso de uso
- Los patrones de inversión de control (como los manejadores de eventos o *inyección de dependencia*) y los patrones de devolución de llamada (se envía un mensaje a un elemento –generalmente un servicio externo que requiere conexión asíncrona– con dirección de respuesta –remitente– para que el servicio conecte con el cliente cuando termine de procesar la petición) ayudan a proteger los elementos de nivel superior de la arquitectura frente a los detalles de implementación de los elementos de nivel inferior

Aplicar estilos arquitectónicos basados en metamodelos

Si tiene requisitos importantes sobre la evolución del sistema, puede valer la pena considerar la adopción de un estilo arquitectónico general que se centre particularmente en apoyar el cambio. Los sistemas basados en metamodelos (o metasistemas) proporcionan un grado muy alto de flexibilidad en algunos dominios problemáticos (particularmente los sistemas de bases de datos que requieren una evolución significativa del esquema).

Los enfoques metamodelo desglosan el procesamiento y los datos del sistema en sus bloques de construcción fundamentales y usan configuraciones de tiempo de ejecución para ensamblarlos en componentes completamente funcionales. Los cambios en los requisitos a menudo se pueden hacer cambiando el metamodelo, en lugar de tener que cambiar los componentes de software subyacentes. Un ejemplo son los sistemas de gestión de contenidos (del inglés [Content Management System \(CMS\)](#)), que se especializan en la creación de blogs y periódicos en línea, wikis, comercio, etc. Cuando se aplican en educación, suelen llamarse herramientas de gestión de contenidos de aprendizaje (del inglés [Learning Content Management System \(LCMS\)](#)), siendo un ejemplo destacado para nosotros Moodle, por ser el sistema en el que se basa la plataforma PRADO de la Universidad de Granada. Otro ejemplo se da a continuación:

EJEMPLO 3.6.18. Uso de un estilo arquitectónico basado en un metamodelo

Un banco de inversión necesita capturar y procesar los detalles de sus productos, las cuales hacen uso de diversos instrumentos financieros (compras de bonos, transacciones de divisas, transacciones del mercado monetario, operaciones de capital, transacciones derivadas, etc.). De forma regular inventan nuevos tipos de productos financieros, lo que significa que el requisito de apoyar la evolución funcional es muy significativo.

Una arquitectura tradicional identificará un cierto número fijo de tipos de productos ofrecidos, implementando funcionalidad para cada uno de ellos e intentando un procesamiento genérico reutilizable para los aspectos comunes. Al aparecer un nuevo producto, habría que cambiar el sistema para darle cabida.

Por el contrario, una arquitectura basada en metamodelos comienza considerando conceptos/entidades fundamentales como clientes (contrapartes), monedas, fechas de negociación y liquidación, límites de negociación, colecciones de productos (libros) para un comerciante en particular, etc. En lugar de desarrollar un sistema para procesar un conjunto particular de tipos de transacciones o productos, el arquitecto diseña un sistema para proporcionar un conjunto de facilidades para implementar los conceptos subyacentes junto con otras de configuración basada en datos para permitir que los implementadores del sistema definan los tipos de productos que desean ofrecer en términos de estos conceptos subyacentes. Más tarde, cuando se requieren nuevos tipos de productos, se agregan cambiando los datos de configuración, en lugar del código del sistema.

La contrapartida de no necesitar reprogramarse sino sólo reconfigurarse es que son mucho más complejos de desarrollar y menos eficientes, lo que puede limitar su aplicabilidad en entornos donde el rendimiento es una preocupación importante.

Construir puntos de cambio en el software

Una estrategia intermedia es adoptar soluciones de diseño que admitan ciertos tipos de cambios en lugares específicos del sistema, lo que requiere identificar los lugares donde puede haber cambios críticos (*puntos de cambio*) y especificar los mecanismos a adoptar para lograrlos.

Además del uso de diversos patrones de diseño que introducen alguna forma de punto de cambio (Fachada, Cadena de Responsabilidad y Puente, etc.) se pueden seguir los siguientes enfoques generales:

- Hacer que los elementos sean reemplazables: generalmente implica que la interfaz a un elemento y su implementación se mantengan separadas para que otros elementos dependan sólo de la interfaz. Esto permite cambiar el comportamiento del sistema reemplazando un elemento en el momento de la compilación o, con algunas tecnologías y lenguajes de programación, en tiempo de ejecución

- Controlar el comportamiento mediante la configuración: por ejemplo, las entradas, salidas y precisión requeridas de un elemento de procesamiento estadístico en el sistema para permitir que algunos aspectos de la operación del sistema cambien con el tiempo sin modificar su implementación itemUtilizar datos autodescriptivos y procesamiento genérico: ciertos tipos de procesamiento, como conversión de formato, a menudo se pueden realizar de una manera más genérica si se conoce la estructura de los datos de entrada, por lo que se recomienda el uso de un flujo de datos autodescriptivo (como XML) de tal manera que se use la estructura de los datos entrantes para guiar el procesamiento (como cambio de formato)
- Separar el procesamiento físico y lógico: útil cuando el formato de datos cambia con frecuencia, pero no la funcionalidad que se necesita hacer con ellos. Si el software procesa el formato físico de los datos y luego realiza el procesamiento lógico sobre los resultados, será mucho más fácil adaptarlo a un cambio en el formato físico
- Dividir los procesos en pasos: podemos introducir un posible punto de cambio si cada paso de un proceso se programa como un elemento separado

Al igual que otras decisiones de arquitectura de software, debemos ser cautos al introducir puntos de cambio, sopesando el coste derivado de la creación y mantenimiento de cada punto de cambio con la probabilidad de que se use y su importancia dentro de las necesidades de las partes interesadas.

Usar puntos de extensión estándar

Un enfoque relacionado con el anterior es construir puntos de extensión dentro de una tecnología estándar que proporcione esta posibilidad gratuita y flexible de permitir evolución en el sistema. Por ejemplo, la plataforma J2EE permite crearlos para añadir de forma fácil soporte a nuevos tipos de bases de datos (a través de la interfaz JDBC), y a sistemas externos (a través de la interfaz JCA).

Para ello podemos construir adaptadores personalizados que nos permitan utilizar facilidades de integración de aplicaciones estándar para conectar con nuestros sistemas internos y con los paquetes que necesitemos usar, evitando la construcción de mecanismos propios de integración en nuestro sistema.

Hacer cambios fiables

Un cambio mal diseñado puede provocar efectos secundarios graves que causen problemas importantes en el sistema en explotación. Se proponen las siguientes estrategias para ayudar a que los cambios sean fiables:

- Gestor de la configuración del software: permite controlar los cambios en los módulos software e identificar y recuperar las distintas versiones del sistema

- Proceso de compilación automatizado: junto con el control de las versiones que serán las entradas al proceso de compilación/linkado/construcción del software, es importante crear un sistema automatizado para este proceso que garantice el mismo resultado para las mismas entradas
- Análisis de dependencias: hay muchas herramientas para automatizar este análisis, y usarlas una vez que comencemos a construir el sistema puede ayudar a resaltar dependencias de las que de otro modo no habríamos estado al tanto
- Proceso de lanzamiento automatizado: crear y mantener sistemas para el lanzamiento automatizado que empaqueten el sistema y lo preparen requiere tiempo y esfuerzo, pero en nuestra experiencia siempre es más barato que la alternativa de hacerlo de forma manual
- Creación de mecanismos para deshacer los lanzamientos fallidos: por muchas pruebas que hagamos puede ocurrir que el lanzamiento falle y debamos volver a una versión anterior, para lo cual debemos asegurarnos de tener alguna forma semiautomática de hacerlo, por ejemplo, scripts para revertir a versiones anteriores de software y deshacer los cambios de estado y modelo de datos introducidos por una versión
- Gestión de la configuración del entorno: también debemos controlar los entornos de desarrollo y producción utilizados para crear y ejecutar el sistema. Estos procesos pueden estar menos respaldados por las herramientas existentes, pero es importante administrar cuidadosamente las versiones exactas de las herramientas de desarrollo y las plataformas de implementación, así como la información de configuración precisa para ellas, para evitar la inestabilidad causada por desajustes entre diferentes entornos
- Pruebas automatizadas: debemos asegurarnos de tener un conjunto completo de pruebas, y de que puedan probarse automáticamente en sistemas grandes, para poder evaluar el impacto de un cambio en el comportamiento del sistema
- Integración continua: para lograr detectar errores en los cambios lo más pronto posible, podemos integrar continuamente las partes cambiantes del sistema en lugar de intentar la integración de todo al final del proceso (integración *big bang*), para ello hay que reunir los cambios del sistema con la mayor frecuencia posible y probar el resultado (al menos una vez al día en la mayoría de los casos)

Preservar los entornos de desarrollo

Una vez que un proyecto ha proporcionado una cantidad significativa de funcionalidad, el entorno de desarrollo original a menudo se desmantela o evoluciona. Con el tiempo, puede llegar fácilmente al punto en el que nadie conoce el conjunto exacto de compiladores, sistemas operativos, parches, bibliotecas, herramientas de compilación, etc., que se utilizan para crear el sistema. Esto puede ser un problema particular para los desarrolladores de productos que admiten una amplia gama de plataformas y versiones de productos.

Parte de la responsabilidad del arquitecto es preservar el entorno de desarrollo (compiladores, sistemas operativos, parches, bibliotecas, herramientas de compilación, etc.) de alguna manera, para que sea posible añadir cualquier nueva funcionalidad con el tiempo. Para ellos podemos registrar claramente los detalles del entorno de desarrollo requerido y asegurarnos de que se conserve suficiente hardware y software para que el entorno se pueda recrear con precisión. Una forma de hacerlo es usar herramientas de virtualización de hardware para crear una imagen autocontenido de todo el entorno de software, que se pueden guardar en el disco y aparecer más tarde exactamente en el mismo estado en que se encontraban cuando se guardaron.

Problemas y errores comunes

- Priorización incorrecta de las dimensiones.- Centrarse en las dimensiones evolutivas incorrectas puede dar como resultado una arquitectura que es más compleja y costosa de construir que las alternativas más simples y, sin embargo, no es particularmente fácil de cambiar cuando es necesario. Debemos por tanto hacer primero un estudio previo para estar seguros de enfocarnos en las dimensiones adecuadas
- Cambios que nunca suceden.- Brindar soporte para cualquier cambio futuro requiere una sobrecarga en términos de diseño, implementación y, a menudo, sobrecarga de tiempo de ejecución, por lo que respaldar una serie de cambios que no suceden puede ser un costo innecesario para su sistema. Debemos por tanto brindar soporte a los cambios que consideremos que serán realmente necesarios
- Impactos de la evolución en criterios críticos de calidad.- Si nos centramos demasiado en el objetivo de flexibilidad podríamos hacer un sistema que sea muy fácil de cambiar pero que no cumpla con una o más propiedades de calidad fundamentales, como el rendimiento o la disponibilidad, o tan complejo que descuide otras propiedades como la seguridad o la internacionalización debido a la falta de tiempo. Por tanto debemos asegurarnos de mantener el equilibrio entre la flexibilidad y los otros criterios de calidad importantes para el sistema
- Dependencia excesiva de hardware o software específico.- Dificultan el cambio. Para evitarlas, debemos evaluar el uso de componentes especializados en la arquitectura y asegurarnos de que los beneficios que aportan superan las barreras que se oponen al cambio. También debemos conocer las hojas de ruta de los proveedores y otros factores que pueden limitar la vida útil de los componentes especializados y abstraer las interfaces que haya a componentes especializados para que podamos intercambiarlos sin demasiado impacto
- Ambientes de desarrollo perdidos.- los entornos de desarrollo a menudo están sujetos a cambios y evolución independientes a medida que pasa el tiempo. El problema al intentar recrear un entorno de desarrollo o prueba es que a menudo no está claro exactamente qué se necesita para hacerlo (versiones de bibliotecas, compiladores, lenguajes

de scripting, parches software, versión de sistema operativo, modelos de componentes hardware, etc.). Para reducir el riesgo, cada vez que se introduce un elemento externo en el entorno de desarrollo, debemos registrar su nombre, versión y origen junto con el motivo de su inclusión

- Gestión de lanzamiento ad hoc.- Es importante organizar y administrar el proceso de administración de versiones con el mismo cuidado que el proceso de creación y prueba del sistema. Para ello debemos invertir en un proceso de lanzamiento automatizado para lograr confiabilidad y repetibilidad

Listas de comprobación

Lista de comprobación para captura de los requisitos:

- ¿Has considerado qué dimensiones evolutivas son más importantes para tu sistema?
- ¿Confías en que has realizado un análisis suficiente para confirmar que tu priorización de las dimensiones evolutivas es válida?
- ¿Has identificado cambios específicos particulares que se requerirán y la magnitud de cada uno?
- ¿Has evaluado la probabilidad de que cada uno de esos cambios sea realmente necesario?

Lista de comprobación para la DA:

- ¿Has realizado una evaluación arquitectónica para establecer si tu arquitectura es lo suficientemente flexible como para satisfacer las necesidades evolutivas del sistema?
- Cuando el cambio es probable, ¿tu diseño arquitectónico contiene el cambio en la medida de lo posible?
- ¿Has considerado elegir un estilo arquitectónico inherentemente orientado al cambio? Si es así, ¿has evaluado los costos de hacerlo?
- ¿Has cambiado los costos de tu apoyo a la evolución por las necesidades del sistema en su conjunto? ¿Alguna propiedad de calidad crítica se ve afectada negativamente por el diseño que has adoptado?
- ¿Has diseñado la arquitectura para acomodar sólo aquellos cambios que crees que serán necesarios?
- ¿Puedes recrear sus entornos de desarrollo y prueba de manera confiable?
- ¿Puedes construir, probar y lanzar de manera confiable y repetible el sistema, incluida la capacidad de revertir los cambios si salen mal?
- ¿Es tu enfoque evolutivo elegido la opción más barata y menos arriesgada de entregar el sistema inicial y la evolución futura requerida?

3.7. Conclusiones

Como conclusión, consideramos como una parte muy importante de esta asignatura, en cuanto a su contribución para ir madurando hacia una mejor formación como ingenieros informáticos, ser capaces de elaborar una DA completa a partir de un supuesto práctico, usando la propuesta de Rozansky⁶⁴ o cualquier otra en la que se tengan en cuenta los criterios de calidad y se permita incorporar a ellos todas las consideraciones, tanto legales como éticas, que sean responsabilidad del ingeniero.

⁶⁴Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*.

Acrónimos

CMS Content Management System

DA Descripción Arquitectónica

LCMS Learning Content Management System

UML Unified Modeling Language

Bibliografía

- Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. <https://learning.oreilly.com/library/view/pattern-oriented-software-architecture/9781118725269/>.
- Gottesdiener, Ellen. «Running a use case/scenario workshop». En *Scenarios, stories, use cases through the systems development life-cycle*, editado por Ian F. Alexander y Maide Mein. Wiley, 2004. <https://www.oreilly.com/library/view/scenarios-stories-use/9780470861943/>.
- Group, IEEE Architecture Working. *IEEE P1471/D5.0 Information Technology - Draft Recommended Practice for Architectural Description*. Informe técnico. 1999. <http://www.pithecanthropus.com/~awg/>.
- Hilliard, Rich. «Using the UML for Architectural Description». Editado por Springer. *Proceedings of UML'99, Lecture Notes in Computer Science 1723* (1999).
- Rozanski, Nick. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*. Addison-Wesley Professional, 2011. <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.
- Shaw, Mary y David Garlan. *Software Architecture*. New Jersey: Prentice Hall, 1996.
- Stafford, Judith, Robert Nord, Paulo Merson, Reed Little, James Ivers, David Garlan, Len Bass, Feliz Bachmann y Paul Clements. *Documenting Software Architectures: Views and Beyond, Second Edition*. EE.UU.: Addison-Wesley Professional, 2010.

Tema 4. Prueba software

Desarrollo de Software

Curso 2022-2023

3º - Grado Ingeniería Informática

Dpto. Lenguajes y Sistemas Informáticos

ETSIIT

Universidad de Granada

12 de mayo de 2023



Tema 4. Prueba software

Contenidos

4.1.	Calidad, garantía de calidad y control de calidad	6
4.1.1.	Prueba estática y prueba dinámica	9
4.1.2.	Prueba y depuración no son la misma tarea	9
4.1.3.	Prueba software o validación (control de calidad), verificación (garantía de calidad) y calificación	10
4.2.	Fundamentos de la prueba software	11
4.2.1.	Mantener el software bajo control	11
4.2.2.	Principios generales de la prueba software	11
4.2.3.	El proceso de la prueba software	13
4.3.	La prueba software en los distintos modelos del ciclo de vida del software	16
4.3.1.	El modelo en cascada	17
4.3.2.	El modelo en espiral o iterativo	17
4.3.3.	El modelo en V	18
4.4.	Clasificación de las pruebas	20
4.4.1.	Pruebas de caja blanca y pruebas de caja negra	20
4.4.2.	Clasificación de las pruebas según los requisitos	21
4.4.3.	Pruebas automatizadas	21
4.4.4.	Pruebas Beta	22
4.5.	Pruebas de unidad	22
4.5.1.	Ejemplo pruebas de unidad en Dart/Flutter	24
4.6.	Pruebas de componentes	25
4.6.1.	Ejemplo de prueba de componentes en Flutter	26
4.7.	Pruebas de integración	28
	Bibliografía	31

Prueba software

En este tema abordaremos la parte del desarrollo del software que se refiere a la prueba software. En la sección 4.1 definiremos los conceptos fundamentales desde donde ubicar el de prueba software, calidad, garantía de calidad y control de calidad, así como otros conceptos relacionados con la calidad y la prueba software. En la sección 4.2 revisaremos los fundamentos de la prueba software. La sección 4.3 la dedicaremos a presentar cómo se integra la prueba en los distintos modelos del ciclo de vida software. Por último, se presentarán varias clasificaciones de pruebas según distintos requisitos, en la sección 4.4.

Para las sección 4.1, salvo otras referencias explícitas, nos basaremos en el libro *Software Testing, Principles and Practices*, de Desikan y Ramesh,¹ pero también usaremos el enfoque más integrador de Daniel Galin en *Software Quality*.² En la sección 4.2, usaremos principalmente el libro *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*³ de Hambling *et al.*. Para la sección 4.3 usaremos sobre todo el libro de Desikan y Ramesh⁴ y para la última sección, la sección 4.4, seguiremos el libro de Daniel Galin.⁵

¹Srinivasan Desikan y Gopalaswamy Ramesh, *Software Testing, Principles and Practices* (India: Pearson; O'Reilly Media, Inc., 2007), <https://learning.oreilly.com/library/view/software-testing-principles/9788177581218/>.

²Daniel Galin, *Software Quality* (Wiley-IEEE Computer Society Press, 2018), <https://learning.oreilly.com/library/view/software-quality/9781119134497/c14.xhtml>.

³Brian Hambling et al., *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*, EBL-Schweitzer (O'Reilly Media, Inc., 2015), <https://learning.oreilly.com/library/view/software-testing/9781780174921/>.

⁴Desikan y Ramesh, *Software Testing, Principles and Practices*.

⁵Galin, *Software Quality*.

4.1. Calidad, garantía de calidad y control de calidad

NOTA: Seguiremos en esta sección principalmente el libro de Desikan y Ramesh,⁶ por ello no lo citaremos constantemente. Solo cuando se usen otras fuentes, las citaremos.

La *prueba software* está relacionada con el concepto de *calidad software*. ¿Pero qué es la calidad? La respuesta ha sufrido cambios a lo largo del tiempo. Véase primero la definición de calidad de Desikan y Ramesh de 2007⁷ (definición 4.1.1) y una definición bastante más completa de IEEE en el estándar IEEE 730-2014⁸ (definición 4.1.2). Debe observarse que en esta segunda definición se considera que para que el software sea de calidad no solo debe limitarse a cumplir con la especificación de requisitos sino que estos requisitos deben responder a las verdaderas intenciones de las partes interesadas.

DEFINICIÓN 4.1.1: Calidad^a

^aDesikan y Ramesh, *Software Testing, Principles and Practices*.

La calidad consiste en cumplir los requisitos que se esperan del software, de forma consistente y predecible.

DEFINICIÓN 4.1.2: Calidad IEEE Std. 730-2014^a

^aIEEE, «IEEE Standard for Software Quality Assurance Processes».

Es el grado en el que un producto software cumple los requisitos establecidos. Sin embargo, la calidad también depende del grado de fidelidad con el que los requisitos establecidos representan las necesidades, deseos y expectativas de las partes interesadas.

Ambas definiciones coinciden en que la calidad está relacionada con el cumplimiento de unos requisitos establecidos. Hay dos métodos de aumentar la calidad del software según se concibe en la primera definición, es decir, que se cumplan sus requerimientos de forma consistente y predecible:

- Control de calidad software.- Se basa en un ciclo de construcción/reconstrucción iterativa de un producto hasta que tenga el comportamiento que se esperaba antes de ser construido. Es por tanto un método orientado al producto (detectar errores y corregirlos) y no al proceso. Existe un equipo específico para el control de calidad. Galin⁹ proporciona una definición similar (definición 4.1.3 1) y una segunda acepción con claras diferencias, que la considera un proceso llevado a cabo por los desarrolladores del producto (definición 4.1.3 2).

⁶Desikan y Ramesh, *Software Testing, Principles and Practices*.

⁷Desikan y Ramesh.

⁸IEEE, «IEEE Standard for Software Quality Assurance Processes», *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, 2014, 1-138, <https://doi.org/10.1109/IEEESTD.2014.6835311>.

⁹Galin, *Software Quality*.

DEFINICIÓN 4.1.3: Control de calidad software

1. Conjunto de actividades destinadas a evaluar la calidad de un producto desarrollado o manufacturado.
2. El proceso de verificar el trabajo de desarrollo propio o el de un compañero de trabajo.

■ Garantía de calidad software.- Se basa en la prevención de errores interviniendo en cada fase del ciclo de vida del producto, antes y después. Por ejemplo, después del diseño, o antes de codificar exigiendo la aplicación del estándares que ayuden a prevenir errores. La responsabilidad recae en todas las personas que intervienen en el desarrollo del producto e incluso en los usuarios finales. Los estándares ISO (como ISO 9000) pretenden especificar la forma más correcta de proceder en distintos contextos, para garantizar la calidad. El estándar IEEE 730-2014 proporciona una definición más amplia (definición 4.1.4) que se ajusta a su también más amplia definición de calidad (ver anterior definición 4.1.2). Por otro lado, Galin¹⁰ amplía aún más este concepto para relacionar también con la garantía de calidad los servicios de operación y el cumplimiento de la temporización y el presupuesto (definición 4.1.5).

DEFINICIÓN 4.1.4: Garantía de calidad software (adaptada de IEEE 730-2014)^a

^aIEEE, «IEEE Standard for Software Quality Assurance Processes».

Conjunto de actividades que definen y valoran la adecuación de un proceso software para producir software cuya calidad responda a la intencionalidad de los procesos para los que fue concebido.

DEFINICIÓN 4.1.5: Garantía de calidad software^a

^aGalin, *Software Quality*.

Conjunto de actividades que definen y valoran la adecuación de un proceso software para producir software cuya calidad responda a la intencionalidad de los procesos y servicios para los que fue concebido y cumpla con los requisitos establecidos de ajuste a la temporización y al presupuesto.

En la actualidad, algunos autores consideran el control de calidad como una parte de la garantía de calidad que se refiere a las actividades que garantizan la calidad del producto software, frente a otras que se centran en garantizar la calidad del proceso software.¹¹ De hecho, la idea de garantía de calidad y la importancia que ésta da a detectar los errores lo antes posible, se encuentra en la base de dos cambios muy importantes relacionados con el

¹⁰Galin.

¹¹Galin, *Software Quality*.

control de calidad:¹²

1. Concepto más amplio de *producto software*.- El producto software al que se refiere el control de calidad no es ya solo el producto final, sino cada subproducto obtenido al terminar cada fase en el ciclo de vida. Por ejemplo, son productos software el documento de especificación de requisitos, el documento del diseño del sistema a alto nivel (descripción arquitectónica), el documento del diseño del sistema a bajo nivel, o el código producido. Si para evaluar cada etapa previa a la codificación se utiliza la *revisión*, la etapa de codificación se evalúa con *pruebas software*.
2. Concepto más amplio de *prueba software*.- La prueba software se extiende de forma que también se prueban subproductos o productos parciales de la fase de codificación (módulos o unidades), lo que ha dado lugar a las *pruebas de unidad* y a las *pruebas de integración*. A esta forma de proceder se la ha llamado *prueba incremental*, mientras que al concepto clásico de probar una vez desarrollado el producto completo se lo llama *prueba big-bang*.

La Tabla 4.1 resume la diferencia entre las dos formas clásicas de aumentar la calidad del software.

Garantía de calidad	Control de calidad
Se concentra en el proceso de producción de un producto	Se concentra en el producto en sí
Pretende prevenir defectos	Pretende tanto prevenir defectos como corregirlos
Se desarrolla durante todo el ciclo de vida del producto	Se desarrolla una vez obtenido el producto ¹³
Es competencia de todo el personal	Es una tarea atribuida a un equipo dedicado a ella ¹⁴
Ejemplos: revisiones y auditorías	Ejemplos: pruebas software a varios niveles

Tabla 4.1: Diferencias entre *garantía de calidad* y *control de calidad* [Fuente:¹⁵].

¹²Galin, *Software Quality*.

¹⁵Brian Hambling et al., *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*, EBL-Schweitzer (O'Reilly Media, Inc., 2015), <https://learning.oreilly.com/library/view/software-testing/9781780174921/>.

¹⁴Sin embargo, debe recordarse la definición más amplia de producto software por influencia del concepto de garantía de calidad, vista anteriormente, que hace que el desarrollo también se extienda a las primeras fases del ciclo de vida del producto.

¹⁵Sin embargo, teniendo en cuenta la segunda acepción de control de calidad dada por Galin [Galin](#) (definición 4.1.3 2), también incluye a los miembros del equipo de desarrollo.

4.1.1. Prueba estática y prueba dinámica

- La *prueba estática* se refiere a detectar errores en el software sin ejecutar código. Estas pruebas pueden realizarse directamente sobre el código (*análisis estático*) o sobre documentos (*revisiones*) para quitar ambigüedades y errores que puedan contener.¹⁶
- La *prueba dinámica* es la que se hace sobre el código, utilizando datos de prueba concretas y ejecutando test concretos.¹⁷

Galin¹⁸ y Hambling,¹⁹ a diferencia de otros autores²⁰, consideran que el concepto de *prueba software* debe referirse de forma exclusiva a la *prueba dinámica*. Así por ejemplo, las *revisiones* no son pruebas software. Nosotros utilizaremos a partir de ahora esta distinción.

4.1.2. Prueba y depuración no son la misma tarea

La mayoría de autores distinguen entre prueba y depuración. Así, no se debe confundir la prueba con la depuración (debugging). Las dos son necesarias, pero tienen funciones distintas. Otros²¹ consideran que todas las actividades que implican ejecución de código para detectar/corregir errores son prueba software (prueba dinámica). Por tanto, para ellos la depuración es un caso específico de prueba software. La realidad es que la depuración es también una actividad que debe situarse dentro del control de calidad software (recuérdese la Definición 4.1.3 2.²²). Veamos las diferencias entre ambas actividades tan necesarias:²³

- *Depuración*.- la usan los desarrolladores para identificar errores y corregirlos, pero no suelen considerar las consecuencias que estas correcciones puedan tener en otras partes del sistema. Es fundamental hacer una depuración eficaz y comprobar la robustez del código antes de pasar a la fase de prueba, para que se puedan realizar pruebas de forma rigurosa.
- *Prueba*.- es una exploración sistemática de un componente o sistema para detectar e informar de fallos, sin incluir la corrección²⁴. La prueba debe comprobar también el efecto de los cambios y las correcciones en otras partes del componente o del sistema, con un número suficiente de test. En sistemas de cierta complejidad, la mayor parte de las pruebas las realiza un equipo de pruebas. Sin embargo, algunas pruebas, las de más bajo nivel o *pruebas de unidad* y las *pruebas de componentes* las deben realizar los desarrolladores y otras, como las *pruebas de integración* pueden realizarla tanto el equipo de desarrollo como el de pruebas.

¹⁶Hambling et al., *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*.

¹⁷Hambling et al.

¹⁸Galin, *Software Quality*, cap. 15.

¹⁹Hambling et al., *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*, cap. 2.

²⁰Cf. Desikan y Ramesh, *Software Testing, Principles and Practices*.

²¹Cf. Galin, *Software Quality*.

²²Galin.

²³Hambling et al., *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*, cap. 2.

²⁴Los fallos detectados serán pasados al desarrollador para que los corrija.

4.1.3. Prueba software o validación (control de calidad), verificación (garantía de calidad) y calificación

En este tema nos centraremos en este concepto más específico de *prueba software* que considera solo a la evaluación de un producto o componente (subproducto) software que requiere ejecución de código, la llamada *prueba dinámica* y que se relaciona especialmente con el concepto de validación y el control de calidad. El concepto de verificación es sobre todo aplicable a actividades relacionadas de forma más común con la garantía de calidad, como revisiones o inspecciones²⁵, es decir, la *prueba estática*. Por último, el concepto de calificación está más unido a la verificación, y se puede llevar a cabo también con revisiones e inspecciones:

- Verificación.- Responde a la pregunta: “¿estamos construyendo un correcto sistema?”. El proceso de evaluar un sistema software o componente, producido en una fase determinada de desarrollo, para determinar si cumple de forma correcta y completa con las condiciones y requisitos especificados al empezar a desarrollarlo. Se trata de examinar la consistencia del código producido con los productos desarrollados en la fase anterior, asumiendo que son correctos –bien por que lo eran desde el principio o porque fueron corregidos-. Es un proceso proactivo y algunos ejemplos de actividades dentro de la verificación incluyen las revisiones de requisitos, de diseño y de código. La verificación se puede considerar un concepto equivalente al de garantía de calidad, que se aplicaba en las distintas fases del desarrollo del producto.
- Validación.- Responde a la pregunta: “¿estamos construyendo el sistema correcto?”. El proceso de evaluar un sistema software o componente, producido en una fase determinada de desarrollo, para determinar si cumple de forma correcta y completa con su propósito original. Se trata así de examinar hasta qué punto el producto satisface la intencionalidad con la que fue concebido por el cliente, aun cuando no se hubiera reflejado bien en las especificaciones. La validación y la prueba (dinámica) se pueden considerar conceptos equivalentes al de control de calidad. La validación más estricta es la que corresponde a la prueba como la última etapa dentro de la fase de diseño del software, posterior a la codificación, y anterior a la fase de mantenimiento.
- Calificación.- El proceso de evaluar un sistema software o componente, producido en una fase determinada de desarrollo, para determinar si cumple de forma correcta y completa con los estándares de codificación y documentación profesional y con la estructura, estilos arquitectónicos, patrones de diseño, procedimientos y normas de buena práctica. Se trata de cuidar los aspectos operativos fundamentales para garantizar el mantenimiento del software. Es también un proceso proactivo y con actividades similares a la verificación para llevarlo a cabo.

²⁵Cf. Galin, *Software Quality*, cap. 14.

4.2. Fundamentos de la prueba software

[NOTA: Esta sección está tomada en su gran mayoría del libro de Hambling *et al.*,²⁶ por ello no lo citaremos constantemente. Solo cuando se usen otras fuentes, las citaremos.]

Es una realidad que el software falla mucho más que las producciones de cualquier otra rama de la ingeniería. Un fallo en el software puede producir daños severos a las personas e incluso la muerte (por ejemplo, si causa un accidente aéreo). También puede afectar a empresas e instituciones, con pérdidas de tiempo o económicas (por ejemplo, un error de facturación), o al entorno (por ejemplo, provocando vertidos químicos).

4.2.1. Mantener el software bajo control

Aunque la solución parece fácil (i.e. más y mejores pruebas software), hay que tener en cuenta algunas consideraciones:

- Prueba exhaustiva.- No existe en términos prácticos: es imposible probar de forma exhaustiva un sistema de una mínima complejidad.
- Prueba y riesgo.- Ya que es imposible la prueba exhaustiva, es siempre importante tener en cuenta los riesgos, consiguiendo un equilibrio adecuado. Por ejemplo, el piloto automático de un avión debe ser probado mucho más que un video juego por las consecuencias mucho mayores que tendría un fallo del mismo.
- Prueba y calidad.- La prueba es para detectar problemas, y facilitar la subsanación de los mismos, es por tanto una parte de las actividades de control de calidad destinadas a que el producto software se lance sin defectos de importancia.
- Priorizar.- Dado que no se puede probar todo, hay que ordenar los tests según la importancia de los mismos en cuanto al riesgo de los fallos que están preparados para detectar, probando requisitos funcionales y no funcionales (teniendo en cuenta el criterio de los usuarios y los contratantes). Así, en el caso de que se parasen las pruebas antes de tiempo, al menos estarían hechas las más importantes.
- Parar de probar.- Para decidir cuándo es seguro parar de probar necesitamos un test objetivo con criterios de finalización (número de pruebas a realizar, niveles tolerables de error, etc.), de forma que las presiones de tiempo no nos lleven a parar antes de tiempo y con un elevado nivel de riesgo.

4.2.2. Principios generales de la prueba software

A partir de las consideraciones realizadas, se han establecido unos principios que deben usar los equipos de prueba software con independencia de las distintas técnicas usadas.

²⁶Hambling et al., *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*.

- La prueba exhaustiva no es posible.- Ya se ha comentado antes. Ni siquiera en los distintos módulos de un programa será posible una prueba exhaustiva, con una cierta complejidad de los mismos.
- Probar un programa es tratar de que falle.- La prueba exitosa muestra la presencia de errores (bugs). Un prueba que no encuentre errores no significa que el código probado no los tenga, sino que la prueba puede no ser capaz de detectarlos.
- Prueba temprana.- Los equipos de prueba no tienen que esperar a que el software esté disponible, cuando antes empiecen, mejor respuesta tendrán ante presiones por posible falta de tiempo. Los tests de validación se pueden diseñar en cuanto se completen las especificaciones de requerimientos. Además se pueden hacer tests estáticos para revisar las posibles ambigüedades o errores en dichas especificaciones, que evitarán pasar a desarrollar código erróneo. La Tabla 4.2 muestra los costes comparativos según la fase donde se detectan los errores. Las metodologías ágiles tienen muy en cuenta este principio y fomentan la incorporación de las pruebas a lo largo de todo el proceso de construcción del software.
- Agrupación de defectos.- Suele ocurrir que los errores no se distribuyen de forma homogénea por todo el código sino que se concentran en unos pocos módulos, por razones de (1) complejidad del sistema, (2) código volátil, (3) efectos de volver a cambiar código, o por la (4) (in)experiencia del equipo de desarrollo. Sin embargo, las pruebas deben contemplar también el resto de los módulos.
- *Prueba de repetición y prueba retroactiva:* Las *pruebas de repetición*, pretenden evitar que errores ya corregidos vuelvan a aparecer por nuevos cambios hechos («Any failed execution must yield a test case, to remain a permanent part of the project's test suite.»²⁷). Por otro lado, persiguen detectar nuevos errores considerando que los desarrolladores suelen fijarse más en las partes que ya han fallado pensando en que el código volverá a ser probado con los mismos test, y descuidar otras partes. Las *pruebas retroactivas* persiguen probar otras partes del software una vez corregida la parte que fallaba, para detectar posibles errores colaterales en los cambios realizados.
- Las pruebas dependen del contexto de la aplicación.- Hay que tener en cuenta los riesgos del software que se desarrolle, y las distintas necesidades, como las de garantizar la seguridad o la confidencialidad, el cumplimiento con tiempos de respuesta en sistemas de tiempo real, etc.
- La falacia de la ausencia de errores.- Que no se conozcan errores no significa que el software esté preparado para su lanzamiento. El caso más extremo es que no se conozcan errores porque no se hayan hecho pruebas.

²⁷Bertrand Meyer, «Seven Principles of Software Testing», ed. por IEEE, *Computer* August 2008 (2008): 99-101.

Fase de detección del error	Coste comparativo
Análisis de requisitos	\$1
Codificación	\$10
Pruebas modulares	\$100
Prueba de sistema	\$1000
Prueba de aceptación del usuario	\$10000
Producción	\$100000

Tabla 4.2: Costes comparativos necesarios para corregir los errores, según la fase donde son detectados [Fuente:²⁸].

En general, se puede decir que el coste que provocan los errores software, incrementa de forma exponencial al tiempo que pasa hasta que son detectados (ver Figura 4.1).

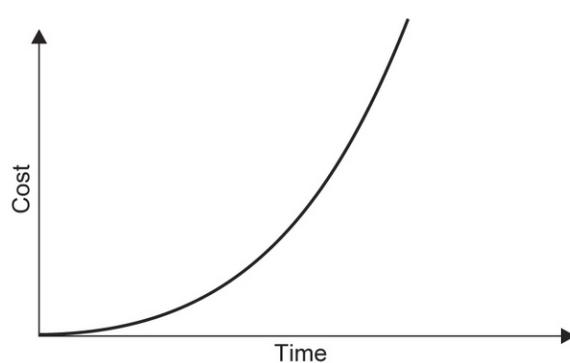


Figura 4.1: Crecimiento exponencial del coste provocado por errores del software según en función del tiempo que pasa hasta su detección [Fuente:²⁹].

4.2.3. El proceso de la prueba software

Prueba como proceso

La prueba software no se reduce solo a pasar los tests: primero hay que establecer los objetivos que se pretenden (por ejemplo, comprobar que un componente cumple con su especificación o intentar encontrar el mayor número de errores de ejecución), y según ellos diseñar las distintas pruebas, el orden adecuado entre ellas, registrar los resultados una vez pasadas las pruebas, enviarlo a los desarrolladores para que sean corregidos los fallos, etc.

²⁸Brian Hambling et al., *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*, EBL-Schweitzer (O'Reilly Media, Inc., 2015), <https://learning.oreilly.com/library/view/software-testing/9781780174921/>.

Las etapas del proceso de prueba

Estas etapas se pueden dividir en 5. A menudo desde una fase hay que volver a otras anteriores, por ejemplo cuando los errores detectados llevan a diseñar nuevos test. Veremos a continuación estas etapas, que se resumen en la Figura 4.2.

1. Planificación y control.- La planificación incluye decidir a grandes rasgos qué se quiere probar y cómo, quién lo hará y cuándo, y qué criterios se usarán para decidir cuándo terminar las pruebas. El control incluye todas las actividades de seguimiento de la planificación y de reajuste entre el plan y la realidad.
2. Análisis y diseño.- A continuación se describen cada una de estas dos subtareas:
 - Análisis de las pruebas.- Consiste en determinar de forma más específica las distintas condiciones a probar. Para ello es necesario realizar dos tareas:
 - Revisar las especificaciones de requisitos, arquitectura, diseño, interfaces y otras partes que deben ser la base para elegir las pruebas.
 - Elegir los elementos a probar y analizar su especificación (v.g. conducta y estructura) para identificar las condiciones y datos requeridos para probarlos.
 - Diseño de las pruebas.- Se trata de especificar cómo combinar estas condiciones de tal forma que impliquen pocos casos de prueba, qué datos se usarán para hacer las pruebas y cuáles son los resultados que un código correcto debe dar en unas condiciones concretas. Se dice que el caso de prueba *pasa* si el comportamiento esperado es exactamente igual al obtenido. De forma más específica, éstas son las tareas que se contemplan en el diseño:
 - Casos de prueba.- A partir de los elementos elegidos para probar y sus condiciones, especificar los casos de prueba a realizar, incluyendo su priorización.
 - Entornos de prueba.- Detallar cómo debe ser cada entorno de prueba y si se requiere alguna infraestructura y herramientas específicas para las pruebas.
 - Datos de prueba.- Especificar los datos de prueba concretos que serán necesarios para los distintos casos de prueba.
 - Trazabilidad pruebas-casos.- Crear una trazabilidad bidireccional entre condiciones base de las pruebas y casos de prueba.
3. Implementación y ejecución.- Esta es la parte más visible, pero requiere de las otras para que tenga utilidad. Es importante tener en cuenta que los casos de prueba deben combinarse en un procedimiento de ejecución global (*escenarios de pruebas* o *colecciones de pruebas* –en inglés, *test suites*– o *grupos de pruebas*) para que la ejecución de las pruebas sea más eficiente. Los resultados deben registrarse, incluidas incidencias, no del código a probar (que es normal) sino del código de la prueba, las cuáles pueden tratarse con excepciones. Además, una vez hechos los cambios hay que realizar tanto *repetición de pruebas* como *pruebas retroactivas*. Todo el proceso de implementación y prueba se puede dividir en las siguientes partes:

- Codificar los procedimientos o métodos para los distintos casos de prueba, crear datos para las prueba y preparar los marcos de prueba automatizados y arneses de prueba (definición 4.5.1) que se necesiten.
- Ejecutar los casos de prueba en un orden determinado, de forma manual o mediante herramientas de ejecución automática de *colecciones de pruebas (suites)*, que ejecutarán los tests en serie para ganar en eficiencia.
- Mantener un registro de las actividades probadas incluyendo:
 - La versión del software probado, los datos, herramientas y el testware (por ejemplo los scripts) usado (ver definición 4.2.1)).
 - Los resultados, comparándolos con los esperados e incluyendo, si es posible, un análisis de causas cuando no se cumple lo esperado:
 - No pasa el test (defectos en el código probado)
 - Fallos en la especificación de la prueba
 - Error en los datos probados
 - Error en la ejecución de la prueba
 - Informar de las diferencias, si las ha habido, distribuyendo este registro.
 - Donde sea necesario, repetir las actividades de prueba cada vez que se cambia el código para corregir los fallos detectados por las pruebas. Esto incluye volver a ejecutar el mismo test (*repetición*), ejecución de un nuevo test que sea correcto y *pruebas de regresión*.

4. Evaluación y reconsideración de los criterios de salida e informe.- En esta fase debemos decidir cuándo parar de probar y comunicar los resultados a las partes interesadas. Se contemplan así las siguientes actividades:

- Comprobar si se ha alcanzado el criterio de parada que se había fijado antes de empezar a ejecutar las pruebas.
- Determinar si se necesita seguir probando o si el criterio fijado debería ser modificado. A veces podemos incluso alcanzar un objetivo fijado (por ejemplo, una cobertura del 85 %), pero surge la necesidad de más pruebas.
- Comunicar a las partes interesadas los resultados finales de las pruebas.

5. Actividades de cierre.- Un vez que se haya decidido terminar con las pruebas en una fase determinada de producto, se debe dar término a un proceso de prueba con las siguientes actividades de cierre:

- Asegurar que la documentación esté en orden y que el producto a entregar esté finalizado, cerrar los incidentes y actualizar los cambios en futuras entregas, documentando que el sistema ha sido aceptado.
- Cerrar y archivar el entorno, infraestructura y el testware usados.
- Pasar el testware al equipo de mantenimiento.

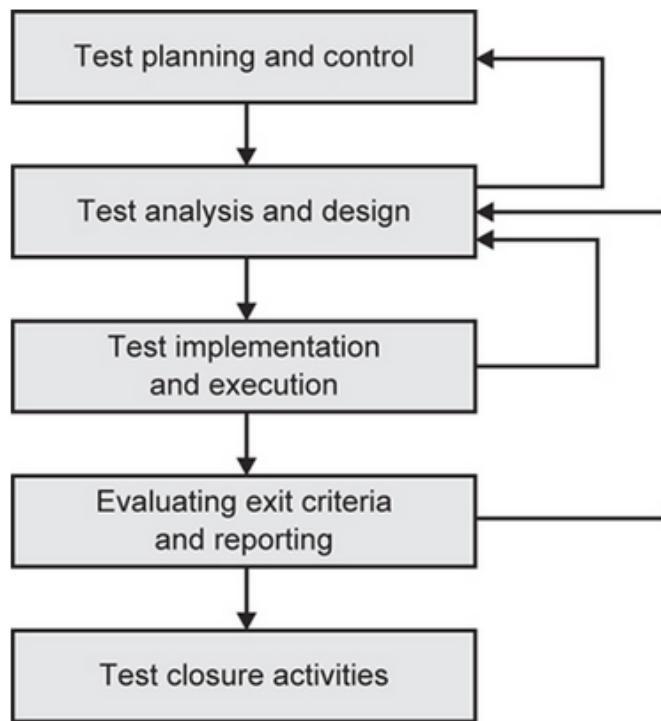


Figura 4.2: Las fases en el proceso de la prueba software [Fuente:³⁰].

- Redactar las lecciones aprendidas con el proyecto de prueba para mejorar los futuros procesos de prueba.

DEFINICIÓN 4.2.1: testware

Colección de software utilizado para la realización de pruebas software, especialmente las que se realizan de forma automática.

4.3. La prueba software en los distintos modelos del ciclo de vida del software

[NOTA: Esta sección está adaptada en su gran mayoría del libro de Desikan y Ramesh,³¹ por ello no lo citaremos constantemente. Solo cuando se usen otras fuentes, las citaremos.]

Los modelos de ciclo de vida describen cómo se combinan las distintas fases necesarias para formar un proyecto o ciclo de vida software completo. Veremos aquí algunos de los más conocidos para fijarnos en qué actividades se realizan en ellos para verificar, validar y calificar el software y, entre ellas destacaremos más especialmente, las actividades de prueba y depuración software.

³¹Desikan y Ramesh, *Software Testing, Principles and Practices*.

4.3.1. El modelo en cascada

El clásico modelo en cascada, realiza las pruebas, incluido el diseño de las mismas, al final de la codificación. El problema de este modelo es que rara vez no hay que ir hacia atrás por un problema encontrado en una fase que requiere modificar la anterior, lo cuál produce altos costes de tiempo y dinero.

La Figura 4.3 muestra un diagrama con el típico modelo en cascada, con las pruebas en último lugar.

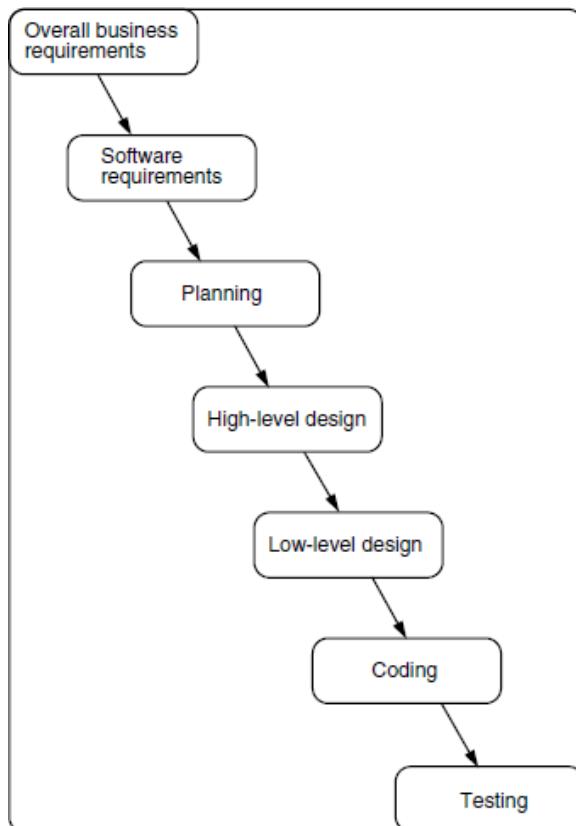


Figura 4.3: Modelo en cascada [Fuente: ³²].

4.3.2. El modelo en espiral o iterativo

El sistema se va desarrollando por requerimientos, de forma que cada requerimiento se puede encontrar en una fase de desarrollo distinta, y por tanto, las pruebas no se realizan todas a la vez. La Figura 4.4 muestra un diagrama con cada requisito en una fase distinta. El R1 en codificación, el R2 en diseño, el R3 en especificación de requisitos, el R4 en prueba y el R5 en explotación.

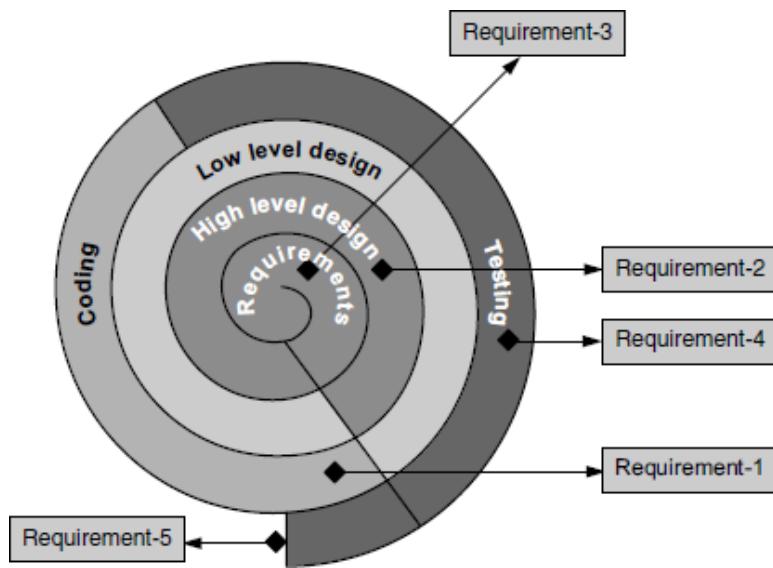
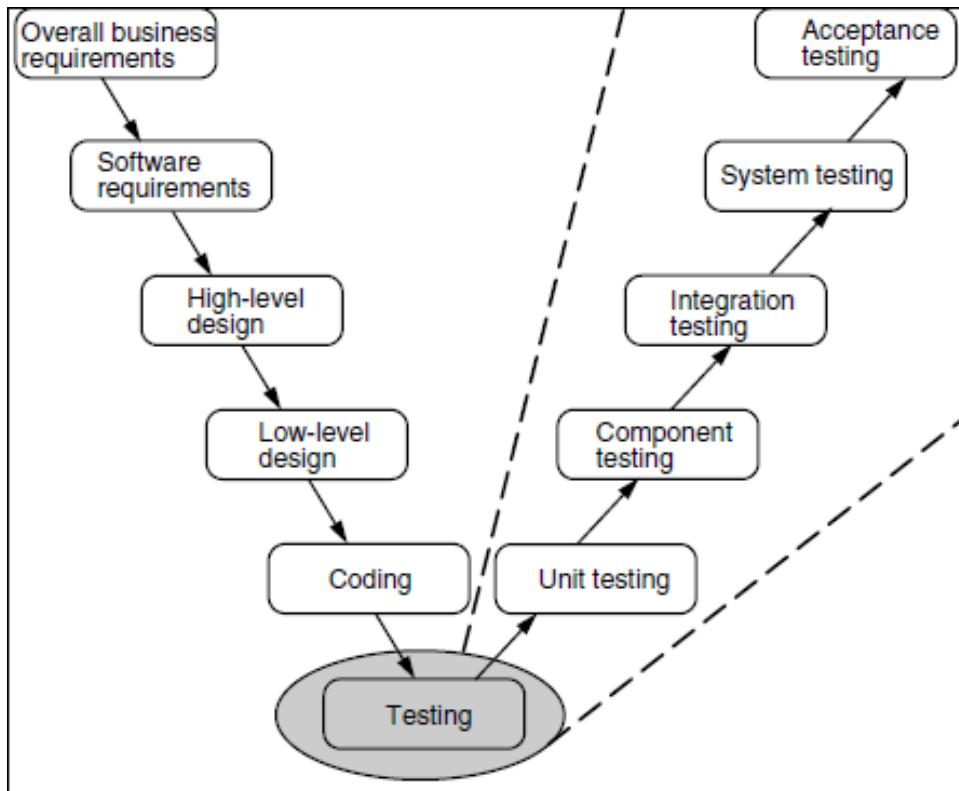
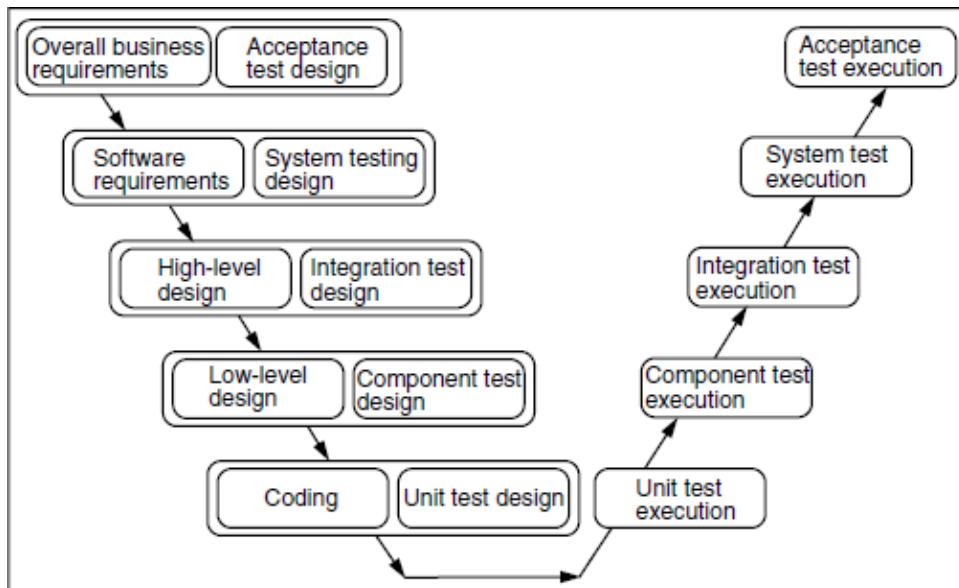


Figura 4.4: Modelo en espiral o iterativo [Fuente:³³].

4.3.3. El modelo en V

Se trata de un modelo parecido al típico modelo en cascada del ciclo de vida software pero que, en vez de considerar la prueba como una fase posterior a la implementación, considera la realización de pruebas en cada fase del producto. La Figura 4.5 muestra el modelo de prueba en V.

En estos modelos, las pruebas se dividen en dos partes: diseño, hecho de forma temprana, y ejecución, realizada al final. Esto se ve más claro en la Figura 4.6.

Figura 4.5: Modelo en V de pruebas [Fuente:³⁴].Figura 4.6: Modelo en V de pruebas [Fuente:³⁵].

4.4. Clasificación de las pruebas

Consideraremos aquí no solo las pruebas dinámicas clásicas o pruebas a secas –que veremos son de *caja negra*– relacionadas con la validación, sino otras pruebas –las de *caja blanca* que incluyen las pruebas estáticas y las dinámicas estructurales (*structural testing*) y que se dirigen a verificar y calificar el código.

4.4.1. Pruebas de caja blanca y pruebas de caja negra

Así, la *prueba de caja negra* encuentra errores de malfuncionamiento del software analizando sus salidas. Aunque las pruebas son mas rápidas de hacer y automatizar, no se identifica la causa del error ni por tanto la calidad del código. Además, unos cuantos errores seguidos pueden dar un resultado correcto que evitaría detectar esos errores.

Las pruebas de caja negra son pruebas que pueden desarrollarse en cualquier nivel, desde pruebas de funcionalidad hasta pruebas de aceptación.

La *prueba de caja blanca* examina las partes internas del procesamiento para identificar errores. Está indicada para detectar errores en el código y verificar su calidad, pero en cambio no puede usarse para realizar algunos tipos de pruebas como pruebas de disponibilidad, confianza, estrés, etc. La figura 4.7 muestra una clasificación de las pruebas de caja blanca.

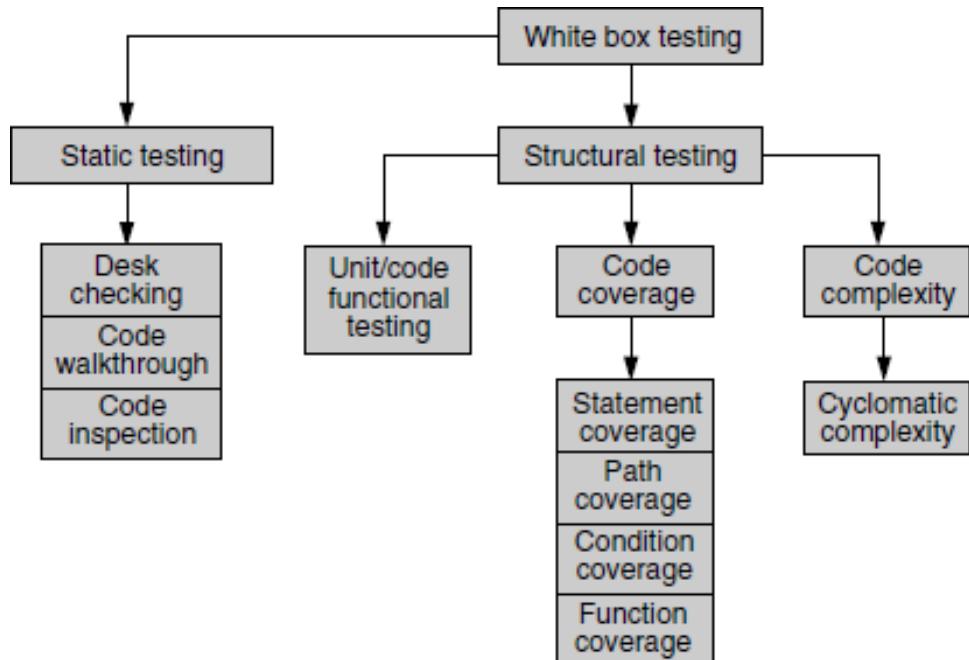


Figura 4.7: Clasificación de las pruebas de caja blanca [Fuente:³⁶].

Existen dos enfoques alternativos para medir la cobertura de los tests de caja blanca:

- Cobertura de camino.- Se define como el porcentaje de caminos posibles que son cubiertos por los casos de prueba. En la mayoría de las situaciones, este enfoque no

es viable dada la cantidad de recursos necesarios para su implementación.

- Cobertura de código/línea.- Se define como el porcentaje de líneas de código que son examinadas durante los casos de prueba.

Las pruebas dinámicas de caja blanca más comunes son pruebas de bajo nivel que suelen desarrollarse como pruebas de unidad o funcionales.

4.4.2. Clasificación de las pruebas según los requisitos

Según el modelo de McCall, las pruebas se clasifican en tres categorías distintas, según los distintos requisitos:

1. Categoría de pruebas según el factor operativo.- Dentro de esta categoría se incluyen los siguientes tipos de pruebas:

- Pruebas de correctitud, incluyendo pruebas de manuales de usuario y pruebas de disponibilidad (tiempo de reacción)
- Pruebas de fiabilidad
- Pruebas de estrés: pruebas de carga y pruebas de durabilidad
- Pruebas de seguridad
- Pruebas de usabilidad: tanto de usabilidad del entrenamiento como de usabilidad operacional

2. Categoría de pruebas según el factor de revisión.- Dentro de esta categoría se incluyen los siguientes tipos de pruebas:

- Pruebas de corrección del mantenimiento
- Pruebas de flexibilidad
- Pruebas de facilidad de testeo

3. Categoría de pruebas según el factor de transición.- Dentro de esta categoría se incluyen los siguientes tipos de pruebas:

- Pruebas de portabilidad
- Pruebas de correctitud del software reutilizado
- Pruebas de interoperabilidad: pruebas de equipamiento e interfaces software

4.4.3. Pruebas automatizadas

Aunque la planificación, el diseño y la preparación de los casos de prueba se hace manualmente, todo lo demás (i.e., pasar los tests, incluidos los de regresión e informar de los resultados, incluyendo informes comparativos) se realiza de forma automática.

Incluyen los siguientes tipos:

- Pruebas de corrección.- Incluyen a su vez:
 - Pruebas de la GUI.- Prueban las distintas formas con las que el usuario selecciona un elemento en una interfaz gráfica: por teclado, click del ratón, tocando la pantalla, etc.
 - Tests funcionales.- Prueban que se realicen las distintas funciones del sistema según aparecen en la especificación de requisitos, detectando las desviaciones de los resultados esperados.
- Pruebas de disponibilidad y de carga.- Los tests deben ser realizados cuando el sistema se encuentre bajo el máximo uso, una condición que suele ser muy difícil de conseguir y es imposible en pruebas manuales. Estos tests permiten establecer el hardware y la configuración necesaria de las comunicaciones para distintos niveles de carga.
- Otros tipos de tests automáticos.- Por ejemplo:
 - Auditorías de código automáticas.- Se trata de usar un auditor automático del código, que prueba si el código se ajusta a estándares y procedimientos especificados de codificación. El informe que produce como salida incluye una lista de desviaciones de los estándares.
 - Monitorización automática de la cobertura.- Se trata de producir informes de la cobertura en líneas de código que se consigue por ficheros de casos de prueba concretos.
 - Pruebas de integridad (seguridad).- Se trata de detectar errores en el software que pueden hacerlo vulnerable a ciberataques. En la actualidad hay herramientas especializadas para este tipo de pruebas.

4.4.4. Pruebas Beta

Las pruebas de sitio Beta consisten en un método por el que se selecciona a un grupo de usuarios, que reciben una versión del software antes de que se ponga en explotación, de forma que lo pueden instalar y probar, informando de los errores que encuentren.

4.5. Pruebas de unidad

Una unidad puede definirse como un trozo de código con un propósito específico. Por ejemplo, una función o procedimiento, y en OO, un método o toda una clase.

Las pruebas de unidad pretenden detectar errores en la lógica interna, probando caminos de control, o en sus estructuras. Son difíciles de mantener, pues al probar el código, cada vez que éste se cambia, hay que cambiar el código de la prueba. Las pruebas de unidad suelen ser consideradas pruebas de caja blanca, pero también pueden ser consideradas de caja negra si probamos la interfaz de la unidad.

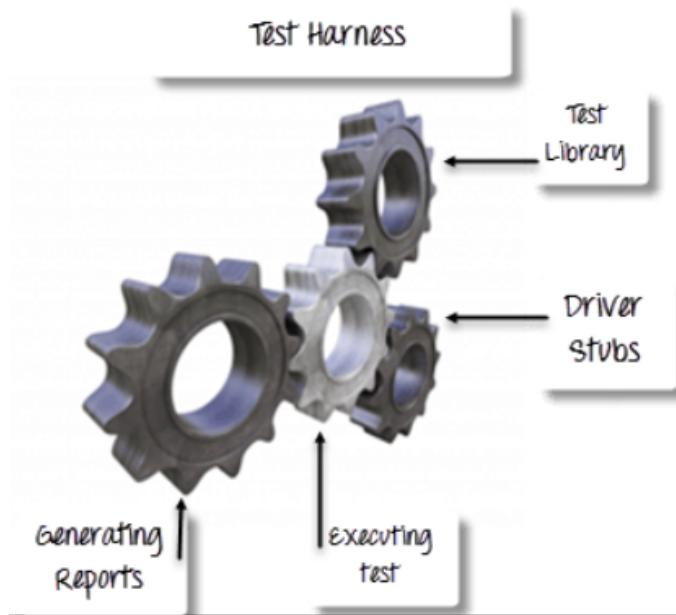


Figura 4.8: Componentes de un arnés de prueba [Fuente: <https://www.guru99.com/what-is-test-harness-comparison.html>].

Para hacer las pruebas de unidad a menudo hay que utilizar *dobles de prueba*, elementos que se hacen pasar por los colaboradores reales de la unidad a probar pero que, en realidad, no lo son.

DEFINICIÓN 4.5.1: Arnés de prueba (test harness) (Figura 4.8)

Colección de software y datos de prueba usados por desarrolladores para probar unidades de software, simulando una pequeña parte del entorno en el que el software funcionará, mediante el uso de controladores (drivers) ficticios (*mocks*) que hagan una verificación dinámica, es decir, prueben el comportamiento (la interacción con otro software –*pruebas de interacción* o *pruebas de caja negra*–) y el uso de objetos ficticios que hagan una verificación estática –*pruebas de unidad* o *pruebas de caja blanca*, es decir, comprueben el estado, silenciando o apagando (*stub out*) toda la parte del código que no se deseé probar (*stubs*). Usan además una librería de pruebas para generar los informes.

Los controladores ficticios (*mocks*) y los silenciadores (*stubs*) –un caso muy simple de mock–, son considerados dos tipos de los llamados *dobles de prueba*.

La Tabla 4.3 resume la diferencia entre controladores ficticios (*mocks*) y silenciadores (*stubs*).

Mocks	Stubs
Son llamados por la clase testeada	Proporciona información a la clase y objeto testeado
Prueban si una clase hace lo que es requerido por una dependencia Verifican el comportamiento	No da detalles sobre cómo la clase usa una dependencia Verifican el estado
Implementan la interfaz y la dependencia	Implementan métodos abstractos de forma que devuelven el valor necesario para hacer la prueba
Pueden provocar un fallo del test	No pueden provocar un fallo del test

Tabla 4.3: Diferencias entre *mocks* y *stubs*.

4.5.1. Ejemplo pruebas de unidad en Dart/Flutter

Consideremos por ejemplo la clase Medallero en Dart:

```
//clase a probar: Medallero
class Medallero {
  static Medallero _instance=null;

  int _totalPremios=0;
  static Medallero getInstance() {
    if (_instance==null)
      _instance=new Medallero();
    return _instance;
  }

  void addPremio() {
    if (_totalPremios < 4)
      _totalPremios++;
    else
      resetPremios();
  }
  void resetPremios()=> _totalPremios=0;

  int get totalPremios => _totalPremios;
}
```

Para crear pruebas de unidad sobre esta clase debemos añadir en la carpeta test un fichero de nombre *medallero_test.dart* y escribir el siguiente código en él. En ese código hay tres pruebas y un grupo o suite de pruebas (*group*) que ejecuta las tres en secuencia. El método clave para hacer cada prueba es el método *expect*, que hará que el test pase si y solo si el primer argumento es igual al segundo.

```
//fichero de prueba: medallero_test
// add package test/test.dart:
// 1: in pubspec.yaml
// add
```

```
//  "dev_dependencies:
//    test:"
// 2: run flutter pub get

import 'package:test/test.dart';
import 'package:flutter_taller_vs/medallero.dart';

void main() {
  group('Medallero', () {
    test('value should start at 0', () {
      final medallero = Medallero.getInstance();
      expect(medallero.totalPremios, 0);
    });

    test('value should be incremented', () {
      final medallero = Medallero.getInstance();
      medallero.addPremio();

      expect(medallero.totalPremios, 1);
    });

    test('value should be reset', () {
      final medallero = Medallero.getInstance();

      medallero.resetPremios();

      expect(medallero.totalPremios, 0);
    });
  });
}
```

4.6. Pruebas de componentes

Un componente es un grupo de unidades relacionadas entre sí a nivel funcional, por ejemplo, un paquete de clases en OO.

Se realizan después de las pruebas de unidad, también por los desarrolladores, pero en este caso con el objetivo de probar el comportamiento y no el código en sí. Es decir, son siempre pruebas de caja negra. Se trata por tanto ya de un comienzo de pruebas orientadas al dominio, a probar la experiencia del usuario, y no al código concreto.

Suelen programarse unidas a la suite o grupo de las pruebas de unidad. A veces reciben el nombre de pruebas de escenarios o características, pues se trata de probar los distintos escenarios o sub-escenarios que representan la interacción del usuario final con ese componente. Ahora no puede haber dobles (stubs o mocks) entre unidades del componente que probamos, pues queremos probar la interacción real entre las distintas unidades de ese componente. Sin embargo, sí que se usarán dobles para silenciar la interacción con unidades de

otros componentes.

4.6.1. Ejemplo de prueba de componentes en Flutter

Veamos un ejemplo en el que queremos comprobar que al pulsar un botón (elemento de la interfaz gráfica de usuario) se incrementa un contador (elemento del modelo).

En el caso de Flutter, a estas pruebas se las llama pruebas de widgets. El código a probar puede ser el de la app que se genera por defecto, la cual tiene solo una página con un botón que incrementa un contador:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);

  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
```

```

appBar: AppBar(
  title: Text(widget.title),
),
body: Center(
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      Text(
        'You have clicked the button this many times:',
      ),
      Text(
        '$_counter',
        style: Theme.of(context).textTheme.headline4
      ),
    ],
),
floatingActionButton: FloatingActionButton(
  onPressed: _incrementCounter,
  tooltip: 'Increment',
  child: Icon(Icons.add),
),
);
}
}
}

```

Igual que con las pruebas de unidad, para probar los widgets, creamos un fichero, por ejemplo *widget_tests.dart* en el directorio test. Implementamos la función *testWidgets*, que crea implícitamente un testeador (*WidgetTester*) para cada prueba a realizar, y, dentro de ella, hacemos uso de distintos métodos de *WidgetTester* para las pruebas: *pumpWidget* que despliega un widget; *tap*, para hacer click sobre un widget botón o *pump* que vuelve a recrear el widget desplegado. A continuación se puede ver un ejemplo de implementación de la prueba. La constante *findsOneWidget* tiene el valor *true* sí y solo si existe un solo widget que cumpla la condición dada al buscador de widgets (clase *Find*) en el primer argumento de *expect*.

```

import 'package:flutter/material.dart';
import 'package:flutter_taller_vs/main.dart';
import 'package:flutter_test/flutter_test.dart';

void main() {
  // Define a test. The TestWidgets function also provides a WidgetTester
  // to work with. The WidgetTester allows you to build and interact
  // with widgets in the test environment.
  testWidgets('MyHomePage button', (WidgetTester tester) async {
    // Create the widget by telling the tester to build it.
    await tester.pumpWidget(MyApp());

    expect(find.text('0'), findsOneWidget);
    expect(find.text('1'), findsNothing);

    // Tap the '+' icon and trigger a frame.
  });
}

```

```
await tester.tap(find.byIcon(Icons.add));
await tester.pump();

// Verify that our counter has incremented.
expect(find.text('0'), findsNothing);
expect(find.text('1'), findsOneWidget);
});

}
```

4.7. Pruebas de integración

Las pruebas de integración tienen el cometido de descubrir errores asociados a la interfaz o interacción entre componentes.

Frente al enfoque big-bang, solo recomendable en sistemas pequeños, el enfoque más usado para probar sistemas grandes cuyos distintos componentes se van terminando de desarrollar en distintos momentos es el *enfoque incremental*.

Existen dos formas de realizar pruebas con un enfoque incremental (ver Figura 4.9):

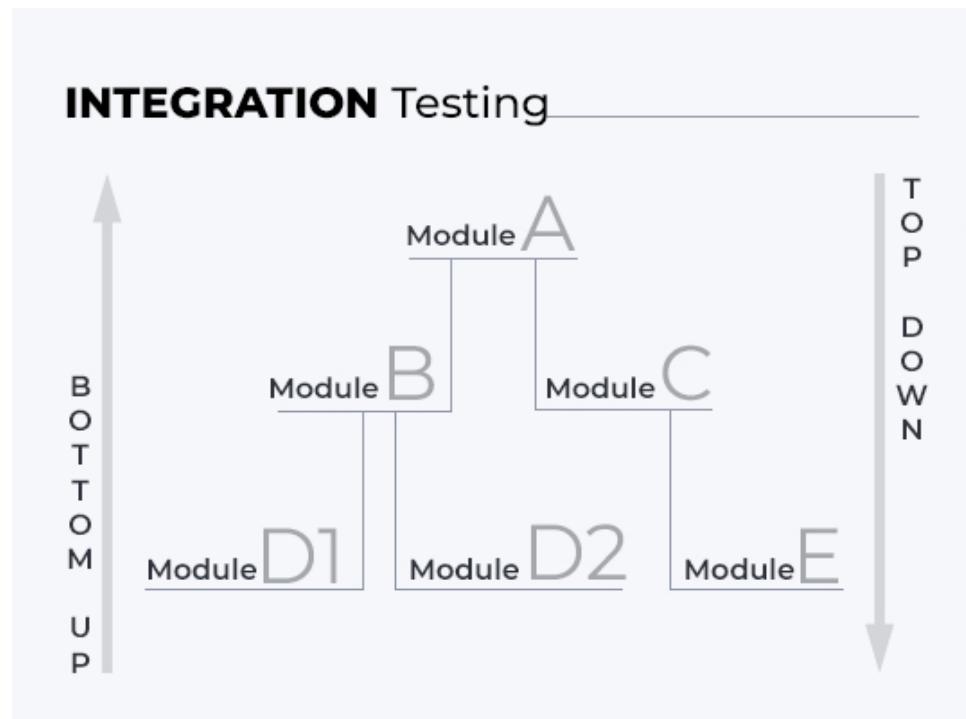


Figura 4.9: Modelo incremental (pruebas de integración) [Fuente:<https://qatestlab.com/services/manual-testing/integration-testing/>].

- Integración top-down.- Se prueban los componentes empezando por el nivel más alto de la jerarquía (programa principal) y terminando por probar los componentes a nivel más bajo. Para probar un componente de un nivel se usan silenciadores (stubs) que reemplazan a los componentes de nivel inferior. Una vez que se prueba un componente, se utiliza dicho componente para probar los de nivel inferior.
- Integración bottom-up.- Se comienza construyendo y probando primero módulos atómicos, es decir, componentes en los niveles más bajos de la estructura del programa, eliminando, de esta forma, la necesidad de crear stubs en la estructura de la prueba de integración.

Bibliografía

- Desikan, Srinivasan y Gopalaswamy Ramesh. *Software Testing, Principles and Practices*. India: Pearson; O'Reilly Media, Inc., 2007. <https://learning.oreilly.com/library/view/software-testing-principles/9788177581218/>.
- Galin, Daniel. *Software Quality*. Wiley-IEEE Computer Society Press, 2018. <https://learning.oreilly.com/library/view/software-quality/9781119134497/c14.xhtml>.
- Hambling, Brian, Peter Morgan, Angelina Samaroo, Geoff Thompson y Peter Williams. *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*. EBL-Schweitzer. O'Reilly Media, Inc., 2015. <https://learning.oreilly.com/library/view/software-testing/9781780174921/>.
- IEEE. «IEEE Standard for Software Quality Assurance Processes». *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, 2014, 1-138. <https://doi.org/10.1109/IEEESTD.2014.6835311>.
- Meyer, Bertrand. «Seven Principles of Software Testing». Editado por IEEE. *Computer* August 2008 (2008): 99-101.

Tema 5. Desarrollo dirigido por modelos

Desarrollo de Software

Curso 2022-2023

3º - Grado Ingeniería Informática

Dpto. Lenguajes y Sistemas Informáticos

ETSIIT

Universidad de Granada

11 de mayo de 2023



Tema 5. Desarrollo dirigido por modelos

Contenidos

5.1.	Fundamentos del desarrollo dirigido por modelos (MDD)	5
5.1.1.	Introducción a la Ingeniería Dirigida por Modelos	6
5.1.2.	Modelos y metamodelos	6
5.1.3.	Lenguaje de modelado (ML)	12
5.1.4.	Productos software, plataformas y transformaciones	15
5.1.5.	De enfoques dirigidos por modelos abstractos a concretos	17
5.2.	Arquitectura dirigida por modelos (MDA)	21
5.2.1.	MDA y algunas extensiones	23
5.2.2.	La propuesta MDA de OMG	24
5.3.	Modelado de procesos de negocio	34
5.3.1.	Modelado y ejecución de procesos de negocio	35
	Acrónimos	47
	Bibliografía	49

Desarrollo dirigido por modelos

El desarrollo dirigido por modelos, también llamada ingeniería dirigida por modelos, es una tendencia de desarrollo software que focaliza el esfuerzo del desarrollo software en modelar bien el dominio del problema (la lógica del mundo que se quiere virtualizar) para poder desarrollar el código a partir de estos modelos. Así, no se limita a modelar el sistema (empezando por el dominio del problema) simplemente como forma de lograr que las distintas partes interesadas tengan la misma imagen del sistema a desarrollar. Comparado con un enfoque “superficial”, que pasa de forma ligera por la fase de análisis, poniendo el mayor esfuerzo en el cómo, en los algoritmos concretos, este enfoque permite construir de forma más rápida sistemas válidos y robustos y facilitará el mantenimiento y evolución del sistema. Sin embargo, a menudo se desarrolla software de forma “superficial”, intentando obtener código rápido a toda costa. Posiblemente el código funcione pero pronto se detectará que no pasa las pruebas de validez (validación) de forma que no cumple con la intencionalidad original con la que el sistema fue concebido por el cliente, y, a veces, ni siquiera es fiable, pues no cumple con algunos requisitos funcionales (verificación). Asimismo, será más difícil poder mantenerlo y que pueda evolucionar. Por otro lado, a partir de los modelos se puede generar código, de forma que hasta el proceso de implementación es más rápido cuando se utiliza un enfoque dirigido por modelos.

5.1. Fundamentos del desarrollo dirigido por modelos (MDD)

En esta sección se expondrá principalmente el trabajo de Alberto Rodrigues da Silva¹ titulado “Model-Driven Engineering: A survey supported by a unified conceptual model”. Para evitar repetidas referencias al mismo trabajo, solo se citarán de forma explícita otros autores, entendiéndose que el texto no citado es una traducción literal o resumida del trabajo de Rodrigues da Silva mencionado.

¹Alberto Rodrigues da Silva, «Model-Driven Engineering: A survey supported by a unified conceptual model», *Computer Languages, Systems & Structures* 20 (junio de 2015), <https://doi.org/10.1016/j.cl.2015.06.001>.

5.1.1. Introducción a la Ingeniería Dirigida por Modelos

Un modelo software representa un aspecto o vista del sistema usando un lenguaje (a menudo gráfico) para describirlo. Por un lado, son utilizados como forma de tener una imagen común del sistema a desarrollar, compartida por todos los interesados en el sistema y que permita estar seguros de que se desarrolla lo que de verdad cada uno imagina. Pero lo más importante es que existe una propuesta de desarrollo software, el enfoque **MDE** (del inglés Model-driven Engineering), que saca mucho más partido de los modelos. En concreto a partir de ellos se pueden crear y ejecutar de forma automática sistemas software, a partir de técnicas complejas tales como meta-modelado, transformación de modelos, generación de código e interpretación de modelos. Algunos ejemplos son:

- Model Driven Architecture (**MDA**), definida por el Object Management Group (**OMG**)
- Factorías software, que unen al uso de modelos el uso de patrones, marcos de trabajo y herramientas
- Ingeniería **DSL**, del inglés Domain Specific Languages

Se proporciona aquí una forma de explicar en qué consiste el enfoque **MDE**, que se basa en el uso de un modelo (valga la redundancia) conceptual unificado. Se trata de un modelo conceptual porque con él se definen los conceptos que son parte del mismo: sistema, modelo, metamodelo, lenguaje de modelado (**ML**, del inglés Modeling Language), transformación, plataforma software y producto software. Por otro lado se dice que es unificado porque dichos conceptos se relacionan entre sí usando un diagrama de clases UML y descripciones complementarias.

5.1.2. Modelos y metamodelos

Modelo

Un modelo es una abstracción de un sistema bajo estudio. La Figura 5.1 muestra la definición de sistema. El modelo es en sí también un sistema, con su propia identidad, complejidad, elementos, relaciones, etc. Si pensamos en un modelo sobre un modelo, debemos considerar que el segundo actúa como sistema bajo estudio y que, por tanto, es un sistema.

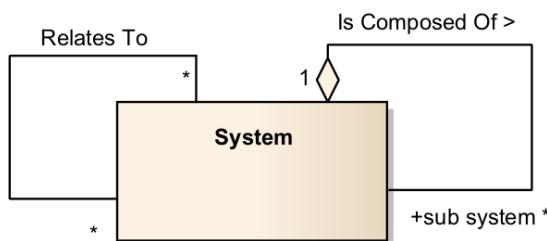


Figura 5.1: Definición de sistema. [Fuente:²]

La Figura 5.2 muestra la definición de modelo y su relación con el sistema que describe.

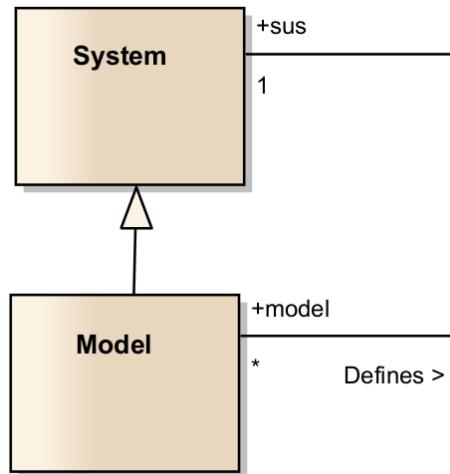


Figura 5.2: Definición de modelo y su relación con el sistema que describe. [Fuente:³]

Como sugiere la figura, podemos definir un modelo de la siguiente forma:

DEFINICIÓN 5.1.1: Modelo

Sistema que ayuda a definir y dar respuesta sobre el sistema bajo estudio sin necesidad de tener que considerarlo directamente

Criterios y principios

Se proponen tres criterios para distinguir un modelo software de cualquier otro artefacto:

- Criterio de mapeo: Debe ser posible identificar en la realidad (si existe) cada objeto representado en el modelo
- Criterio de reducción: El modelo simplifica la realidad, destacando solo algunos aspectos de ella que interesan (también pueden amplificarse otros)
- Criterio de pragmatismo: El modelo debe ser útil para algún propósito, por ejemplo, para Booch los modelos sirven para:
 - visualizar un sistema, como es o queremos que sea;
 - especificar la estructura y el comportamiento de un sistema;
 - funcionar como plantilla que guía el proceso de desarrollo; y
 - ayudar a documentar las decisiones tomadas a lo largo del ciclo de vida de un proyecto

Metamodelo

DEFINICIÓN 5.1.2: Metamodelo

Modelo que define la estructura de un [ML](#).

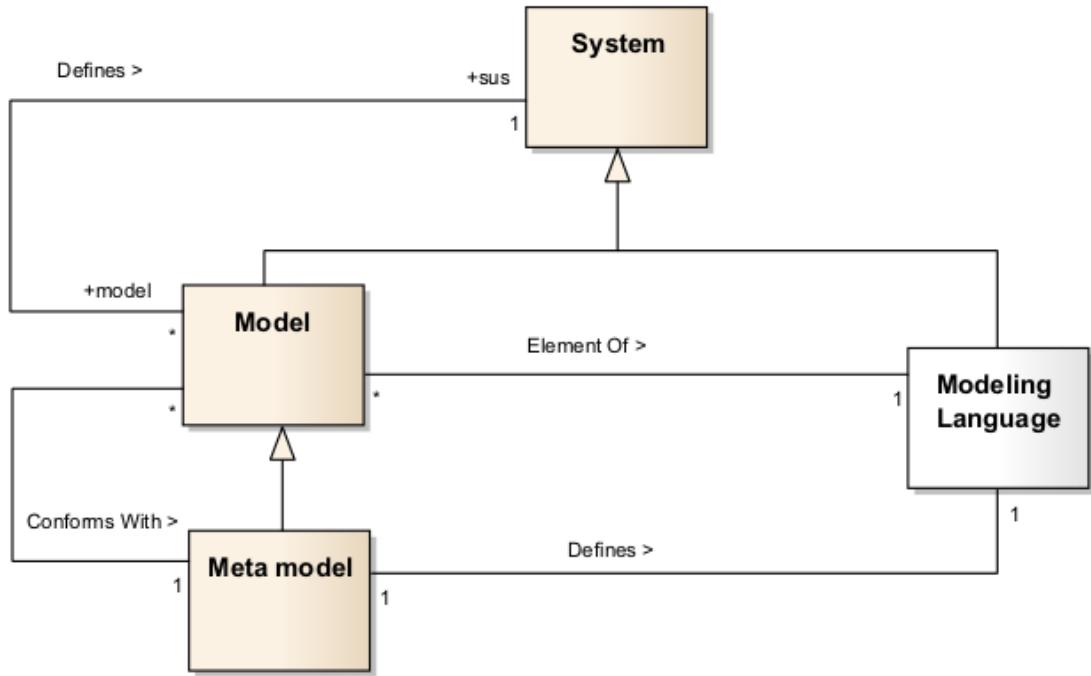


Figura 5.3: Definición de metamodelo y su relación con el modelo. [Fuente:⁴]

Si observamos la Figura 5.3 se cumple que:

1. Relación *ElementOf*: Un **ML** es un conjunto de modelos (o un modelo es un elemento de un **ML**)
2. Relación *Defines*: Un metamodelo es un modelo de la estructura de un **ML** (o un **ML** es definido por un metamodelo)
3. Un metamodelo es un modelo de un conjunto de modelos o es un modelo de modelos
4. Relación *ConformsWith*: Un modelo se ajusta a un metamodelo, i.e. debe satisfacer las reglas definidas al nivel de su metamodelo

Meta-metamodelo, metamodelo y modelo

Un problema conocido y recurrente del metamodelado es cómo configurar el metamodelo inicial. Si un metamodelo es un modelo de un **ML**, debe haber a su vez un meta-metamodelo que describa el **ML** para crear ese metamodelo, y así sucesivamente, en niveles superiores y meta-metamodelos más abstractos. La solución común para superar este problema es utilizar un lenguaje que, en un determinado nivel de esta jerarquía, se describe a sí mismo en su propio lenguaje. Existen numerosos ejemplos de esta solución. Los lenguajes naturales, en especial las lenguas vivas, suelen describir las gramáticas y diccionarios usando el propio lenguaje que se describe. Entre los lenguajes de programación, Lisp es un lenguaje que se describe a sí mismo, de forma que el compilador Lisp está escrito en Lisp.

EJEMPLO 5.1.1. Metaclasses como ejemplos de metamodelos

En el caso de los lenguajes OO, tenemos un ejemplo de modelo en el concepto de clase. La clase es un modelo de los objetos (sistemas) que se pueden instanciar a partir de ella. En lenguajes OO puros, tales como Ruby o Smalltalk, todo es un objeto y también, por tanto, lo son las clases. Esto significa que en tiempo de ejecución se pueden cambiar las propiedades de una clase y por tanto la de sus instancias, añadiendo/eliminando/modificando métodos o atributos. ¿Cuál es la clase de esos “objetos clase”? Es lo que en Smalltalk se llama metaclass. A su vez, ¿quién modela una metaclass? Es decir, ¿cuál es la metaclass de una metaclass? Esto en Smalltalk se resuelve con la autodescripción arriba mencionada. Así, la metaclass de una metaclass es la propia metaclass (ver la parte izquierda de la Figura 5.4).

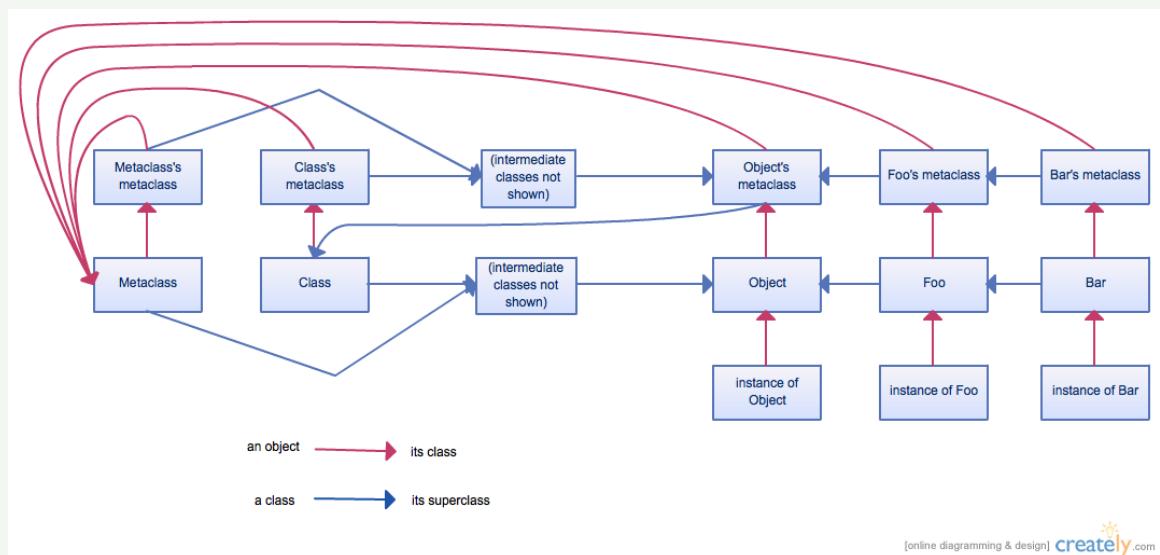


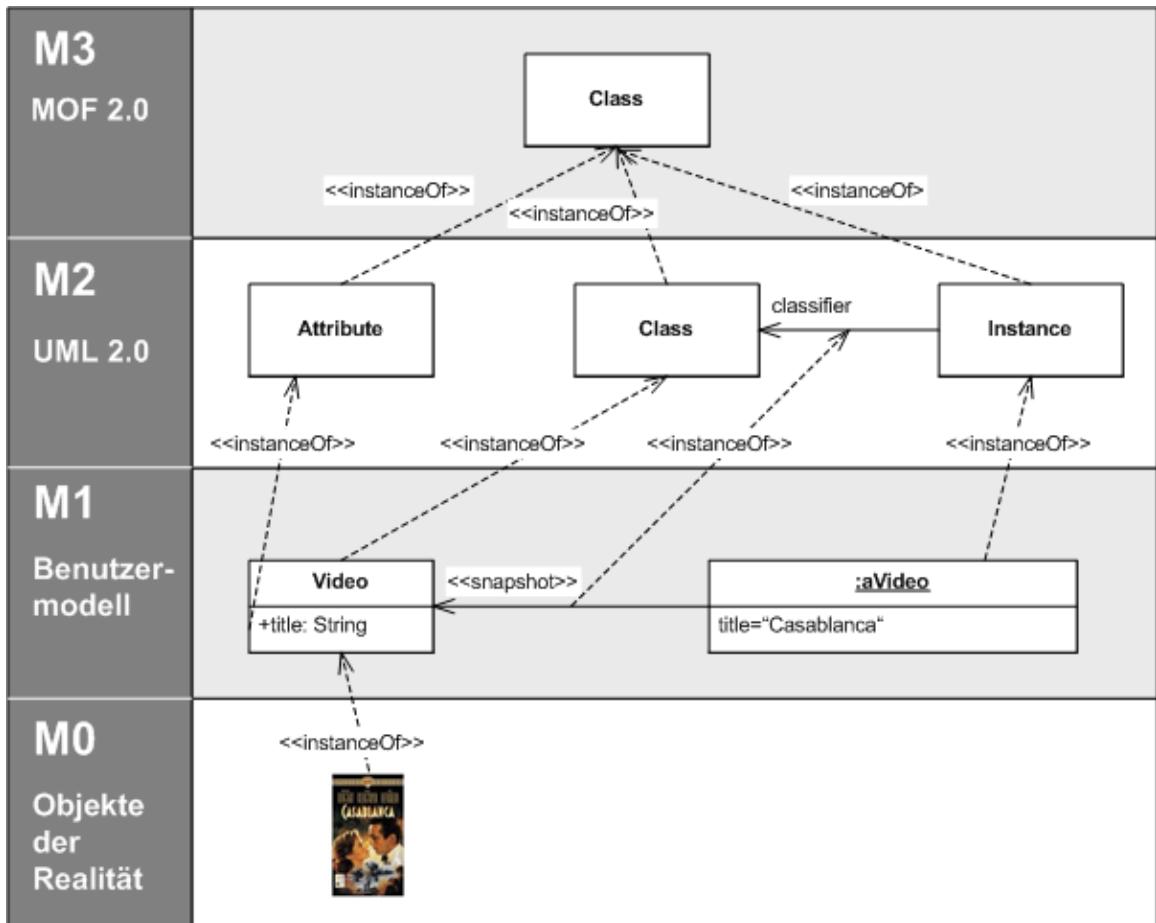
Figura 5.4: Sistema de clases, y relaciones de herencia e instantiación en Smalltalk.
[Fuente: [Wikiwand: “Metaclass”](#)]

En el campo de los **MLs**, un ejemplo es el estándar Meta Object Facility (**MOF**), el núcleo de **MDA** (de **OMG**), el cual asigna tipos (e interfaces para usarlos), a las distintas entidades de una arquitectura. **MOF** se define a sí mismo usando **MOF**.

MOF está formado por una arquitectura de cuatro capas, capas que se describen a continuación de la más alta a la más baja (ver Figura 5.5):

- M3: Un meta-metamodelo, el lenguaje que usa **MOF** para especificar metamodelos (o modelos M2). **MOF** se crea a sí mismo instanciándose de su propio modelo
- M2: Metamodelos, instanciados a partir del meta-metamodelo M3, siendo cada elemento del metamodelo una instancia de un elemento definido en el meta-metamodelo M3. Un ejemplo de metamodelo (es decir, una instancia de **MOF**) es UML, y otro es **Common Warehouse Metamodel (CWM)**

- M1: Modelos, definidos según los intereses y necesidades de sus usuarios (distintos dominios de aplicación, distintos niveles de abstracción) y descritos mediante los metamodelos en M2. Ejemplos sería todos los modelos escritos en UML. Un modelo de usuario puede contener tanto elementos de modelo (como clases –Video, por ejemplo–) o instancias de las clases –instance:Video, por ejemplo–.
- M0: Datos, que describen los objetos que existen en un entorno de cómputo concreto o incluso en el mundo real (un Video concreto en tu portátil).

Figura 5.5: Un ejemplo de la jerarquía de 4 niveles de [MOF](#).

La Figura 5.6 muestra otro ejemplo de la jerarquía de 4 niveles del estándar [MOF](#).

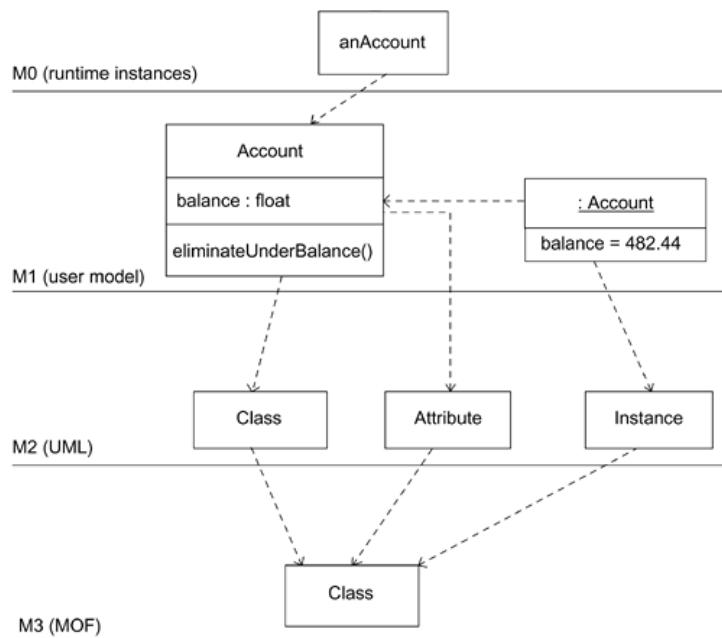


Figura 5.6: Otro ejemplo de la jerarquía de 4 niveles de MOF. [Fuente:⁵]

¿Por qué usar metamodelos?

Mellor et. al. dan tres razones fundamentales para aventurarse en la “Meta-Zona”:⁶

- Especificación del lenguaje:

“Puesto que los modelos deben ser consensuados por todas las partes interesadas en un producto software, necesitamos una forma de especificar exactamente lo que significan los distintos elementos de un modelo concreto y todo el modelo en conjunto. De esta manera, todas las partes interesadas lo puedan entender, y rebatirlo, si consideran que no se ajusta a su idea del sistema que quiere implementarse. Los metamodelos hacen esto precisamente: especifican los conceptos, es decir, el significado de los distintos elementos, que forman parte del lenguaje que está detrás de un modelo concreto.”⁷

- Comunicación acerca de los modelos:

“De esta forma, los metamodelos simplifican la comunicación entre las distintas partes interesadas en un modelo. En lugar de decir “Un *Cliente* es algo que puede tener atributos y operaciones y cuyas instancias potencialmente viven tanto tiempo como el sistema software”, simplemente podemos

⁶Stephen J. Mellor et al., *MDA distilled: Principles of Model-Driven Architecture* (USA: Addison-Wesley Longman Publishing Co., Inc., 2004).

⁷Mellor et al.

decir ”el *Cliente* es una instancia de *Entidad*” o incluso simplemente ”la entidad *Cliente*” y consultar el metamodelo apropiado que nos dice exactamente qué es una *Entidad* (algo que puede tener atributos y operaciones y cuyas instancias potencialmente viven mientras el sistema de software viva, presumiblemente).”⁸

- Funciones de mapeo:

“Además, las funciones de mapeo entre modelos se pueden establecer de manera nítida utilizando conceptos definidos en el metamodelo. Por ejemplo, una función de mapeo podría indicar que un elemento del modelo de origen, como *Class*, se corresponde con un elemento del modelo de destino, como *Entity*. Las ideas clave aquí son que el significado de clase se define en un metamodelo y que el significado de entidad también se define en un metamodelo (diferente).”⁹

5.1.3. Lenguaje de modelado (ML)

Aunque ya se ha definido antes como un conjunto de modelos que se ajustan al metamodelo con el que se relaciona, se da ahora una definición más minuciosa:

DEFINICIÓN 5.1.3: Lenguaje de modelado (ML)

Un ML con una sintaxis abstracta o metamodelo m , una sintaxis concreta o notación n o más, una semántica s , y una pragmática p es un conjunto de todos los posibles modelos que se ajustan al metamodelo m , se representan por una notación n o más, satisfacen la semántica s y tienen en p una guía de la forma de uso más apropiada (ver Figura 5.7).

⁸Mellor et al., *MDA distilled: Principles of Model-Driven Architecture*.

⁹Mellor et al.

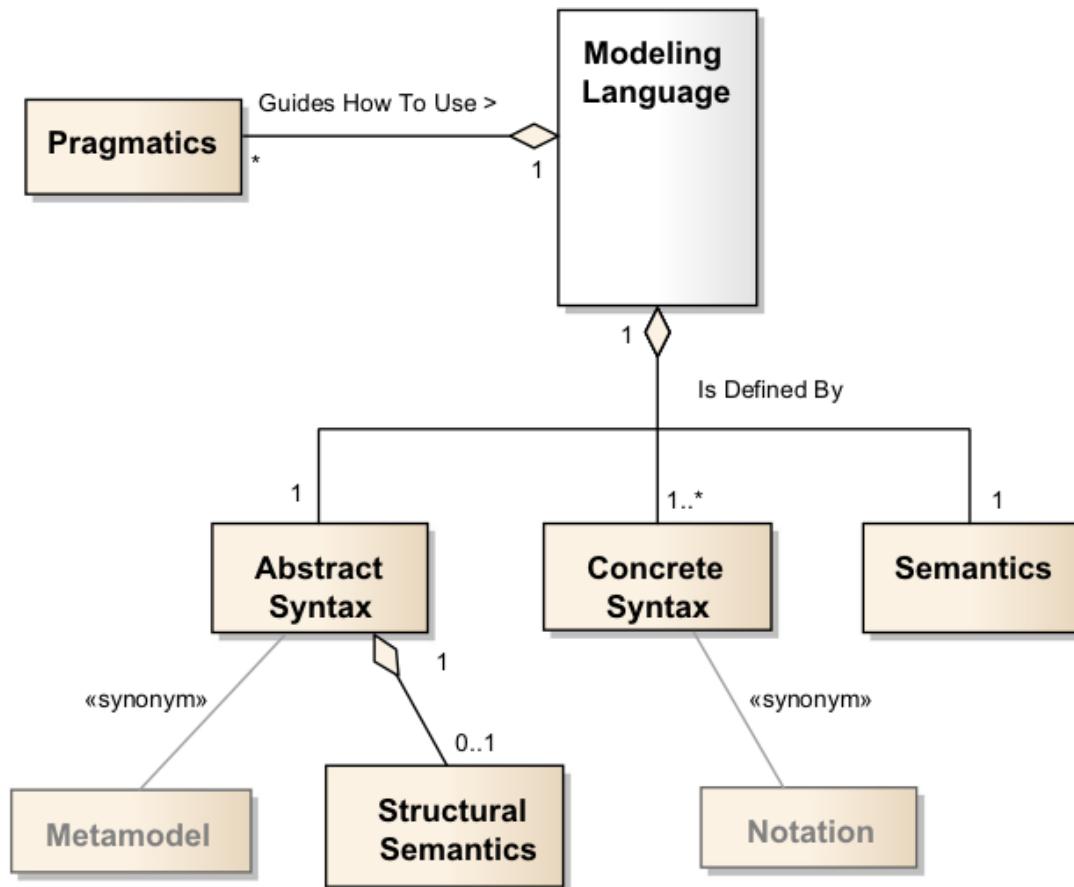


Figura 5.7: Definición de lenguaje de modelado. [Fuente:¹⁰]

Clasificación de un lenguaje de modelado

Un criterio para clasificar a los lenguajes de modelado es el dominio de la aplicación:

- **GPM_L**: de uso general (del inglés, General Purpose Modeling Language). Tienen un número mayor de constructos o conceptos generales, de forma que invitan a un uso más amplio en diferentes campos de aplicación. Ej. UML, SysML¹¹.
- **D(S)ML**: de uso específico (del inglés, Domain-Specific Modeling Language). Usan un número más reducido de constructos y además más relacionados con el campo específico de su dominio de aplicación. Esa mayor especificidad los hace más fáciles de entender y, por tanto, de validar. Sin embargo también tiene mayor costo el que sea específico de un dominio, pues un lenguaje tiene que aprenderse, implementarse, mantenerse, y aprender las herramientas que permiten desarrollo de software usando dicho lenguaje.

¹¹SysML es un lenguaje usado para generar documentación en sistemas software OO.

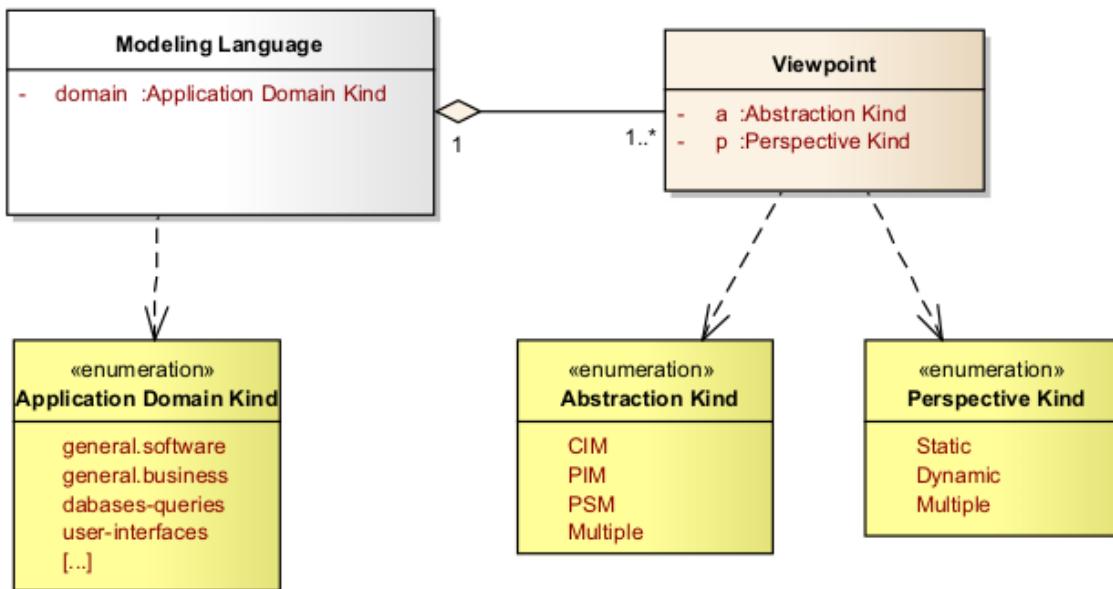
Puntos de vista como formas de estructuración de un lenguaje de modelado

Además los lenguajes de modelado pueden estructurarse en base a distintos puntos de vista, que permiten definir un conjunto de criterios reusables para la construcción, selección y presentación de una parte del modelo que da respuesta a las inquietudes de una parte interesada particular.

Los puntos de vista que proporciona un lenguaje de modelado pueden clasificarse en base a dos criterios:

- Nivel de abstracción (terminología [MDA](#)):
 - Punto de vista independiente de la computación: [Computation Independent Model \(CIM\)](#))
 - Punto de vista independiente de la plataforma: [Platform Independent Model \(PIM\)](#)
 - Punto de vista específico de una plataforma: [Platform Specific Model \(PSM\)](#)
 - Punto de vista múltiple: incluye distintos niveles de abstracción ([CIM](#), [PIM](#), [PSM](#))
- Perspectiva o dimensión funcional:
 - Punto de vista estático: el lenguaje permite describir el sistema desde una perspectiva funcional estructural (clases, objetos, nodos, bloques, relaciones entre ellos). Ej. diagrama de clases UML o diagrama de componentes
 - Punto de vista dinámico: el lenguaje permite describir la funcionalidad del sistema en base al comportamiento del mismo (tareas, operaciones, estados, eventos, mensajes y sus relaciones). Ej. diagramas de actividad o máquinas de transición de estados en UML; diagramas de procesos de negocio en [Business Process Model & Notation \(BPMN\)](#)

La Figura [5.8](#) muestra los distintos tipos de lenguajes de modelado (según el dominio de la aplicación) y sus distintas parcelaciones o puntos de vista, en base a dos criterios para obtener dichos puntos de vista, el nivel de abstracción y la perspectiva o dimensión funcional.

Figura 5.8: Definición de lenguaje de modelado. [Fuente: [12](#)]

5.1.4. Productos software, plataformas y transformaciones

El enfoque **MDE** defiende el uso de lenguajes de modelado tanto para especificar modelos con cierto nivel de abstracción como para ayudar al proceso de desarrollo de un producto software.

DEFINICIÓN 5.1.4: Producto software

Sistema compuesto de la integración no trivial de

- plataformas software,
- artefactos (código) generados por transformaciones modelo-a-texto,
- artefactos (código) escritos directamente por desarrolladores,
- eventualmente, modelos directamente ejecutables en una plataforma software particular.

La Figure 5.9 muestra la relación de un producto software con las plataformas, transformaciones y modelos.

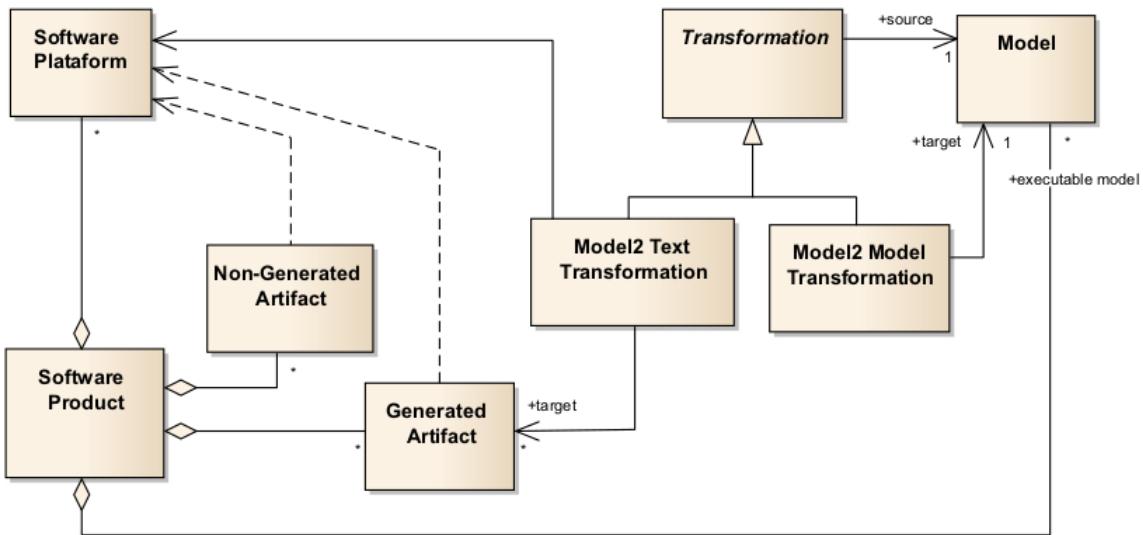


Figura 5.9: Producto software, plataformas, transformaciones y modelos. [Fuente:¹³]

Plataformas software Una plataforma software es un conjunto integrado de elementos computacionales que permiten el desarrollo y ejecución de una clase de productos software. Por lo general, estos elementos proporcionan diferentes funcionalidades a través de la reutilización y mecanismos de extensibilidad y se refieren a tecnologías como middleware, librerías software, marcos de aplicaciones, y componentes de software, pero también sistemas de gestión de bases de datos, servidores Web, gestores de contenidos, sistemas de gestión de documentos, sistemas de gestión de flujo de trabajo, etc.

Artefactos Los artefactos, tanto generados de forma automática desde el modelo como escritos por desarrolladores, pueden ser relevantes durante el tiempo de desarrollo, durante el tiempo de ejecución o en ambas fases del ciclo de vida del producto software. En todo caso, todos dependen estrechamente de las plataformas involucradas. Algunos ejemplos son: archivos de código fuente y binario, scripts de configuración e implementación, scripts de bases de datos e incluso archivos de documentación, incluidos los propios modelos.

Transformaciones En el enfoque MDE, se consideran dos tipos de transformaciones:

- Transformaciones de modelo a texto (**M2T**, del inglés Model To Text).- Producen artefactos software textuales –generalmente código, XML y otros ficheros de texto– a partir de modelos. La técnica más común dentro de la transformación **M2T** es la de generación de código
 - Transformaciones de modelo a modelo (**M2M**, del inglés Model to Model).- Transforman un modelo en otro, siendo generalmente el segundo más cercano al dominio de la solución o satisfaciendo las necesidades de alguna parte interesada. Estas transformaciones se especifican a través de distintos lenguajes, como lenguajes de programación,

pero también por lenguajes especializados de transformación de modelos, para diferentes propósitos y con diferentes paradigmas de modelado (algunos ejemplos son [QVT](#), Acceleo, ATL, VIATRA y DSLTrans)

Modelos Los modelos – que se crean manualmente pero que también se pueden producir automáticamente a partir de transformaciones de [M2M](#) y, a continuación, pudiéndose editar y refinarse) — se utilizan para producir artefactos de software – también manualmente (no generados) o mediante transformaciones [M2T](#) (generados).

En algunas situaciones particulares, los modelos pueden ser directamente interpretados y ejecutados por plataformas específicas integradas con el producto software. Para ello deben definirse de manera consistente y rigurosa. En general, se requiere un cierto nivel de calidad para que los modelos se puedan usar correctamente en transformaciones [M2M](#) o [M2T](#).

5.1.5. De enfoques dirigidos por modelos abstractos a concretos

Los enfoques [MDE](#) pueden clasificarse en tres tipos según el nivel de abstracción:

- Enfoques de alto nivel
- Enfoques de nivel medio: meta-herramientas dirigidas por modelos
- Enfoques concretos: herramientas dirigidas por modelos

La Figura 5.10 muestra una taxonomía de [MDE](#), con distintos niveles de abstracción.

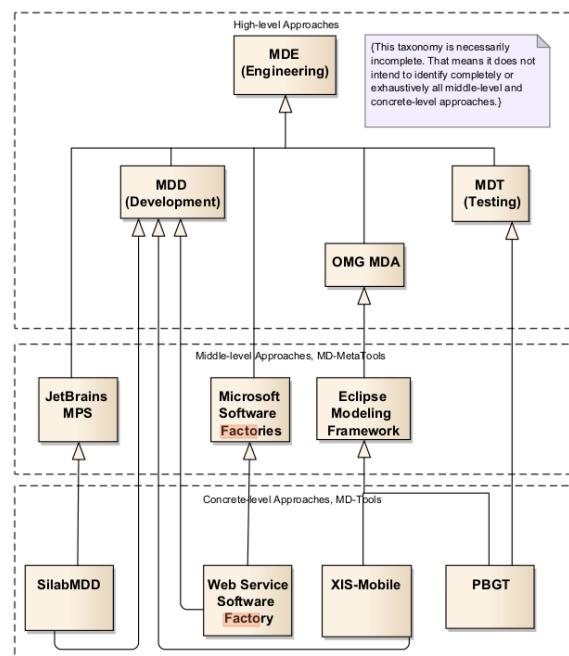


Figura 5.10: [MDE](#) y distintos niveles de cocreación. [Fuente:¹⁴]

Enfoques de alto nivel

A este nivel se muestran tres tipos de **MDE**:

- Orientados al desarrollo (**MDD**, del inglés Model Driven Development).- Se enfocan en las tareas de especificación de requisitos, análisis, diseño e implementación. Las concreciones de **MDD** suelen proporcionar lenguajes para modelar con distintos niveles de abstracción y transformaciones **M2M** and **M2T** para mejorar tanto productividad en el desarrollo como calidad en el producto software generado
- Orientados a la prueba (**MDT** o **MBT**, del inglés Model Driven-Based Testing).- Se enfocan en la automatización de las pruebas. Las concreciones **MDT** suelen representar el comportamiento que se desea del sistema, las estrategias de prueba y el entorno de prueba. Los casos de prueba generados están al mismo nivel de abstracción que el modelo y podrán convertirse en tests ejecutables que conectan con el sistema usando unas herramientas de prueba y marcos de trabajo concretos
- **MDA**, de **OMG**.- Es el enfoque **MD** propuesto por el **OMG**, enfocado principalmente en la definición de modelos y sus transformaciones. Admite la definición de modelos en diferentes niveles de abstracción (**CIM**, **PIM**, **PSM**). Las plataformas computacionales corresponden a implementaciones concretas de servidores de aplicaciones, servidores de bases de datos, gestores de contenidos, frameworks y arquitecturas software; estas plataformas se pueden describir a través de Modelos de Descripción de Plataforma (**PDM**, del inglés Platform Description Model). También considera diferentes tipos de transformaciones **M2M**: **CIM-CIM**, **CIM-PIM**, **PIM-PIM**, **PIM-PSM** y **PSM-PSM**. Además, considera la transformación **M2T** de modelos **PSM** (es decir, **PSM-text**), tanto en código fuente como en otros tipos de artefactos textuales. Gracias en especial a las transformaciones **PIM-PSM PSM-PSM** y **PSM-Text**, una aplicación desarrollada bajo el enfoque **MDA** puede instalarse en diferentes plataformas informáticas y admite adecuadamente diferentes tecnologías. Aunque no especifica lenguajes de modelado concretos ni herramientas asociadas, en la Figura 5.10 aparece un poco más abajo que **MDD** y **MBT** porque defiende el uso de varias especificaciones concretas de **OMG** (se verán más adelante en la Sección 5.2.2)

Enfoques de nivel medio (**MD MetaTools**)

En el nivel medio de la Figura 5.10 identificamos los enfoques **MD** propuestos por sus respectivas corporaciones o comunidades, conducidos por tecnologías y generalmente respaldados por herramientas complejas que llamamos “Metaherramientas **MD**” (o “workbench de lenguajes”). La mayoría de estas herramientas proporcionan una colección de características para ayudar a los usuarios a definir DS(M)Ls, con editores específicos, validación del modelo, transformación del modelo, etc. Aunque existen muchos, se dan solo tres ejemplos (de la comunidad Eclipse, de Microsoft y software libre):

- Eclipse Modeling Project (EMP).- Se centra en la evolución y promoción de tecnologías de desarrollo **MD** dentro de la comunidad Eclipse, proporcionando un conjunto

integrado de marcos y herramientas de modelado extensibles e implementaciones de estándares, con el marco de modelado Eclipse ([EMF](#), Eclipse Modeling Framework) en el núcleo, herramientas de modelado gráfico, textual y concreto (compatibles con las especificaciones [OMG](#)) tales como UML, [OCL](#), SysML y [BPMN](#). Hay varias herramientas y marcos desarrollados sobre [EMF](#), la mayoría populares, relativamente fáciles de usar y mantener y con soporte comunitario abierto y fuerte.

- Microsoft Software Factories.- Se inspira en la metáfora que hace de las “línea de montaje” en áreas de automatización industrial de las fábricas.

DEFINICIÓN 5.1.5: Fábrica de software

Colección estructurada de activos software relacionados, tales como procesos, DSLs, plantillas, IDEs, configuraciones y vistas, que son utilizados para crear tipos específicos de software

Un ejemplo de conjunto integrado de herramientas para crear factorías software es el “Visualization and Modeling [SDK](#)” (antes se llamaban herramientas [DSL](#)) de Microsoft Visual Studio. Permite en particular ayudar a definir DSLs, con su respectivos generadores automáticos de código fuente y de documentación.

- JetBrains Meta Programming System ([MPS](#)).- De código abierto, es un workbench de proyección para lenguajes, lo que significa que no existen ni gramáticas ni parsers, sino que se cambia directamente el árbol subyacente de sintaxis abstracta, mostrado como texto, con un simple editor. Admite notaciones mixtas (texto, símbolos, tablas, gráficos) y una amplia gama de características de composición de lenguajes basadas en su meta metamodelo (llamado BaseLanguage). Los usuarios de [MPS](#) amplían este BaseLanguage para definir sus propios lenguajes, derivando directamente conceptos de BaseLanguage o combinando conceptos de otros lenguajes existentes.

Enfoques de nivel concreto ([MD tools](#))

Por último, la parte inferior de la figura [5.10](#) muestra algunos ejemplos de enfoques [MD](#) concretos, que muestran cómo aplicar [MD](#):

- XIS-Mobile (del inglés, eXtensible Interactive Systems): Enfoque [MDD](#) para aumentar la productividad del desarrollo de aplicaciones móviles multiplataforma. Su [DSL](#) se define como perfil UML, con una arquitectura de múltiples vistas que permite un enfoque básico de diseño y otro inteligente. Tiene además un marco de soporte basado en la tecnología [MD](#) de Enterprise Architect (Sparx Systems) – la llamada MDG (Model Driven Generation)– y [EMF](#), para generar código fuente para múltiples plataformas a partir de una sola especificación de modelo [PIM](#), a través de transformaciones [M2M](#) y [M2T](#). Compuesto por cuatro componentes principales, este marco sugiere desarrollar una aplicación móvil en cuatro pasos siempre que sea posible:

1. Definir las vistas requeridas usando el editor gráfico (Visual Editor),

2. validarlas con el validador de modelos,
3. generar los modelos de las vistas de las interfaces de usuario, usando el generador de modelos y
4. generar el código fuente de la aplicación a través del generador de código.

De esta manera, el desarrollador aprovecha los beneficios de [MDD](#), de mejorar su productividad utilizando una sola especificación del sistema, evitando la implementación de código repetitivo y reduciendo errores.

- WebService Software Factory, o Service Factory, es un ejemplo de las fábricas de software de Microsoft y un enfoque [MDD](#) concreto que proporciona una colección integrada de recursos diseñados para construir servicios Web de forma rápida y consistente, que se adhieran a patrones de arquitectura y diseño bien conocidos. Estos recursos consisten en una guía escrita sobre patrones y cuestiones de arquitectura y herramientas integradas en Visual Studio para especificar modelos y generar código a partir de ellos.
- SilabMDD es un enfoque [MDD](#) enfocado especialmente en la especificación de requisitos (enfoque MDRE (del inglés Model-driven Requirement Engineering). Incluye el lenguaje SilabReq, –implementado con JetBrains [MPS](#)–, un [DSL](#) textual que permite a los usuarios definir y gestionar requisitos mediante la especificación de casos de uso. De esta forma, SilabReq impone una definición rigurosa de especificación de casos de uso, basada en la descripción de secuencias de acciones, pre- y post-condiciones, y las relaciones entre los casos de uso y los elementos definidos en los modelos de dominio del sistema (especificados de forma textual). El objetivo de SilabMDD es proporcionar un workbench para desarrollo completo de software (extendiendo [MPS](#), de JetBrains) para ser utilizado por ingenieros de requisitos y arquitectos software, desarrolladores y partes interesadas no técnicas.
- GTGT (Pattern Based GUI Testing) es un enfoque [MBT](#) que proporciona estrategias de prueba genéricas, basadas en patrones de prueba de interfaz de usuario (UI, del inglés User Interface), con múltiples configuraciones para probar diferentes implementaciones de patrones UI. Es compatible con la herramienta del mismo nombre, en la cual los patrones UI de prueba se definen dentro de un lenguaje específico de dominio, PARADIGM, desarrollado sobre el [EMF](#) usando el meta-metamodelo Ecore. La herramienta PBGT está disponible gratuitamente como un plugin Eclipse. Es un entorno de prueba totalmente integrado que proporciona funcionalidades para el modelado (manual o automático), configuración, generación y ejecución automática de casos de prueba y análisis de cobertura de prueba. Actualmente la herramienta PBGT puede probar tanto aplicaciones Web como apps para móviles Android, sobre el framework Selenium.

5.2. Arquitectura dirigida por modelos (MDA)

La Figura 5.11 muestra el ciclo de vida tradicional en la elaboración de un producto software hasta su despliegue. El problema principal de este enfoque, como muestra la figura, es que el proceso iterativo a nivel práctico se reduce de forma significativa con respecto al teórico. Así, cada vez que se detecta un fallo del software, en vez de volver a la especificación de requisitos o al análisis, si es necesario porque el fallo se haya producido por un error en la propia concepción del sistema, directamente los programadores, que conocen bien el código, se olvidan de la documentación (en forma de diagramas y texto) y cambian directamente el código.¹⁵

Así, la documentación no parece servir ya de mucho. Si el equipo de mantenimiento fuera distinto al equipo de desarrollo, algo que generalmente ocurre en algún momento de la fase de explotación del sistema, tendrían que volver a la documentación para entender el código y ésta estaría incompleta.¹⁶

La realidad es que para los desarrolladores resulta siempre tediosa la elaboración de la documentación, mucho más si se hace con mucho detalle, con una Descripción Arquitectónica (DA) elaborada teniendo en cuenta los puntos de vista de los distintos interesados y las distintas perspectivas para asegurar la calidad,¹⁷ como por ejemplo la propuesta de Rozansky¹⁸ que vimos en el tema 2.

¹⁵Jos Warmer, Anneke Kleppe y Wim Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise* (EE.UU.: Addison-Wesley Professional, 2003).

¹⁶Warmer, Kleppe y Bast.

¹⁷Warmer, Kleppe y Bast.

¹⁸Nick Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition* (Addison-Wesley Professional, 2011), <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.

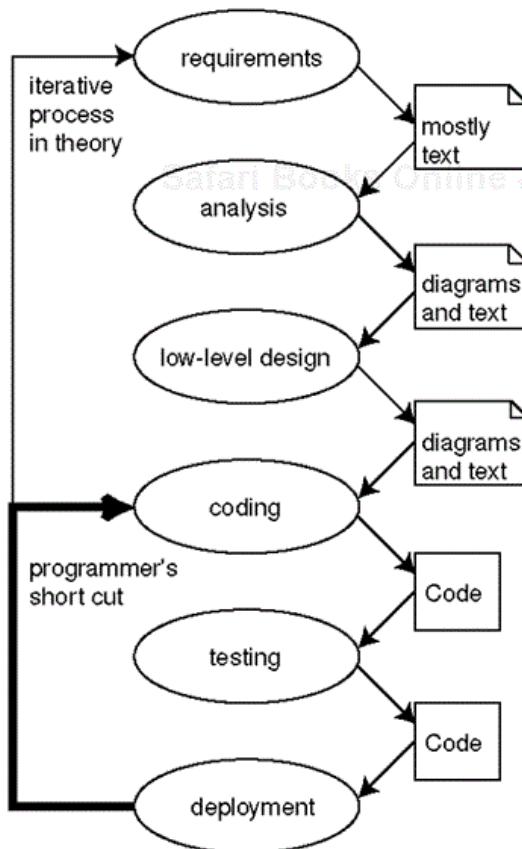


Figura 5.11: El ciclo de vida software tradicional: teoría y práctica. [Fuente:¹⁹]

Las metodologías ágiles, como Scrum o Extreme Programming ([XP](#)), conscientes de esta realidad, la resuelven de una forma “extrema”, de manera que deciden prescindir en la mayor medida posible de las fases iniciales y la documentación generada en ellas y enfocarse sobre todo en las fases de codificación y prueba, para lo que solo usan diagramas de diseño (de bajo nivel). La solución que proponen en el caso de que cambie el equipo de desarrollo o mantenimiento del producto software para no “perderse en el código” es la de dejar “marcadores” para los que vengan detrás, que finalmente son también texto y diagramas de más alto nivel.²⁰

Es decir, en ninguno de los enfoques se puede prescindir realmente de la documentación del alto nivel (la que describe a todo el sistema en su conjunto y en sus partes con un alto nivel de abstracción). De esta forma, no es posible ser siempre realmente productivos, en el sentido de construir algo ejecutable, como es el código.²¹

Sin embargo y como ya hemos visto en la sección anterior, el enfoque [MD](#) plantea una alternativa distinta, en la que desde las primeras fases del ciclo de vida lo que se construye (modelos, expresados por texto y diagramas, principalmente) puedan ya ser productivos,

²⁰Warmer, Kleppe y Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*.

²¹Warmer, Kleppe y Bast.

en el sentido de que puedan ser ejecutados.²²

5.2.1. MDA y algunas extensiones

MDA, la propuesta de (semi) alto MD del OMG (ver Figura 5.10), siendo una propuesta general o de alto nivel, da algunas especificaciones de cómo llevar a cabo el enfoque MD. Una de sus especificaciones es usar un subconjunto de UML como lenguaje de modelado. Una propuesta de nivel medio en cuanto a su nivel de especificación, que no aparece en la Figura 5.10, y que merece la pena mencionar por su amplio uso es xUML (del inglés, executable UML). Existen numerosas herramientas software que implementan xUML, por ejemplo, xtUML (en Bridgeport) y Moka (en Papyrus) –ambas con licencia Eclipse EPL–, Cameo (extensión de MagicDraw), IBM Rational Software Architect Simulation y Cassandra –de pago–, por citar algunas.

La Figura 5.12 resume estas otras extensiones MDA y muestra MDA como una propuesta dentro del enfoque genérico MDE. Obsérvese la relación entre la Figura 5.12 y la Figura 5.10.

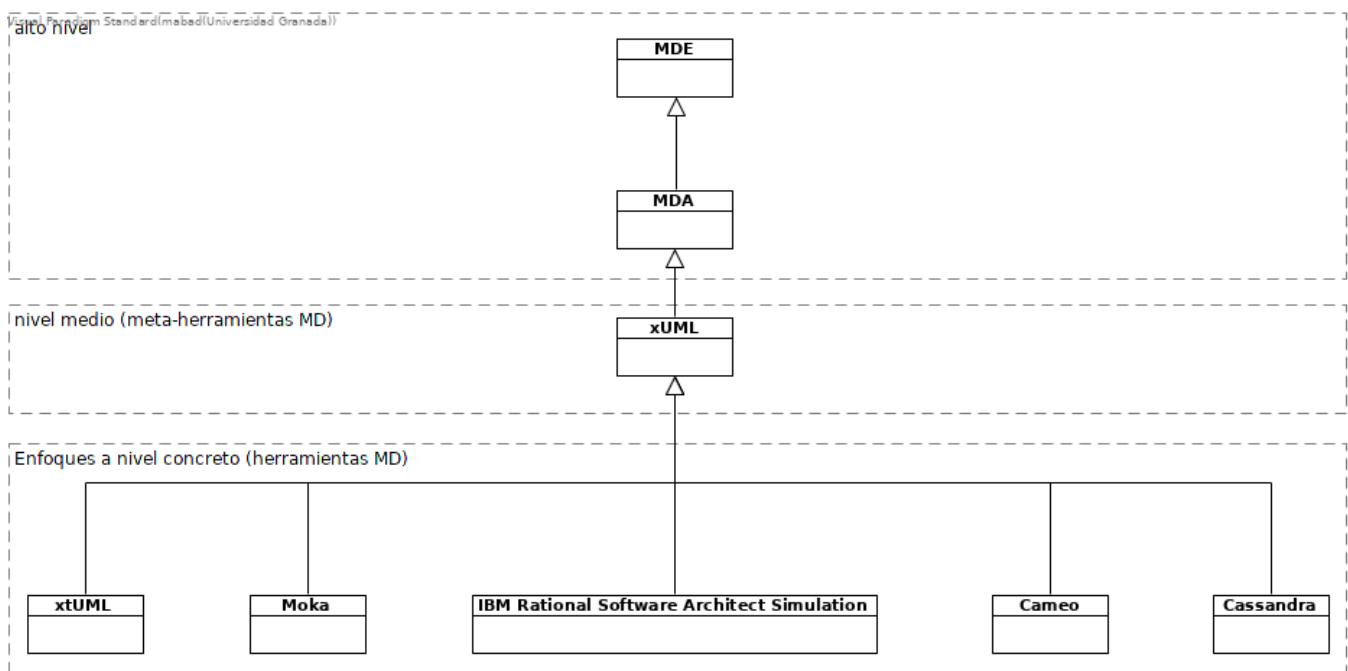


Figura 5.12: Otras extensiones a la propuesta MDA del OMG; y su relación con el enfoque genérico MDE.

CIM, PIM, PSM Antes de ver con más detalle MDA, debemos aclarar que solo expondremos lo que se refiere a PIMs y PSMs. Los CIM son más conocidos como modelos de negocio, los cuales no tienen nada que ver con los sistemas software²³ sino con el objeto

²²Warmer, Kleppe y Bast.

²³En principio, pero veremos más adelante, en la Sección 5.3 que los modelos de negocio también pueden traducirse, en teoría, en código ejecutable.

social de la empresa en sí, la actividad que realiza.²⁴ Por eso se les llama también **CIM**. En la Figura 5.13 puede verse la relación entre un **CIM** (“Business Model”), un **PIM** o un **PSM** (“Software Model”), la empresa (“Business System”) y el software.²⁵

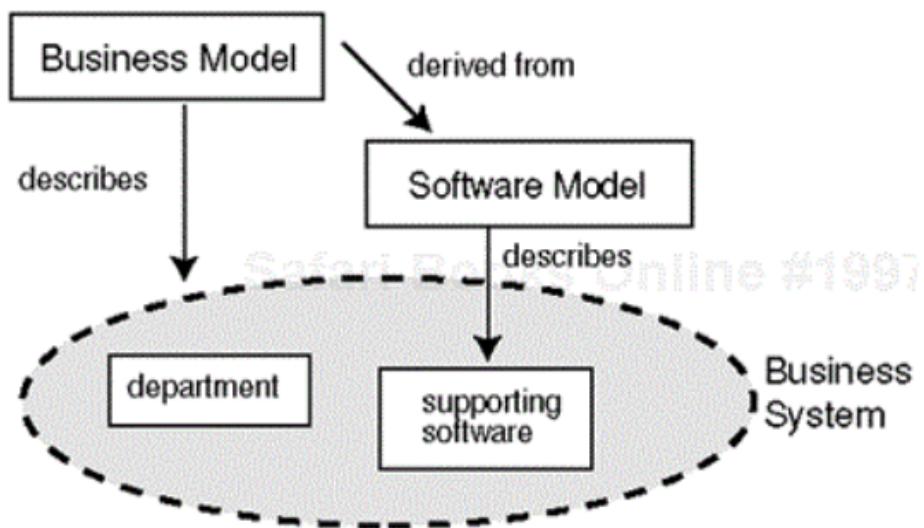


Figura 5.13: Modelo de negocios y modelo software. [Fuente:²⁶]

Para Warmer et. al.,²⁷ y en contra de lo que parece deducirse del trabajo de Rodrigues da Silva²⁸ (véase Sección 5.1.5), el paso de un **CIM** a un **PIM** no es posible automatizarlo, pues la decisión de las partes del **CIM** que se desean plasmar en un producto software debe tomarla el hombre.

Veremos ahora con más detalle **MDA** (Sección 5.2.2).

5.2.2. La propuesta MDA de OMG

Esta sección ha sido extraída (muchas veces con traducción literal) del libro “**MDA Explained: The Model Driven Architecture: Practice and Promise**” de Jos Warmer et. al..²⁹

Para evitar continuas citas, solamente citaremos los contenidos no extraídos de dicho libro.

La Figura 5.14 muestra el ciclo de vida software bajo el enfoque **MDA**. Las diferencias más importantes con el ciclo de vida clásico (teórico y real) de la Figura 5.11 son:

- Teoría y práctica convergen: en efecto, puesto que ahora el propio modelo es ejecutable, cada vez que se necesita cambiar el producto software, se cambiará desde el modelo (análisis), que se volverá a ejecutar produciéndose una nueva versión del producto software.

²⁴Warmer, Kleppe y Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*.

²⁵Warmer, Kleppe y Bast.

²⁷Warmer, Kleppe y Bast.

²⁸Rodrigues da Silva, «Model-Driven Engineering: A survey supported by a unified conceptual model».

²⁹Warmer, Kleppe y Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*.

- Modelos ejecutables: los “productos” o resultados de las fases de análisis y diseño no son modelos expresados en meros diagramas y texto sino modelos perfectamente especificados, en su nivel correspondiente, para poder ser ejecutados o transformados en otros modelos. En concreto, en la fase de análisis se genera un **PIM** y en la fase de diseño se genera un **PSM**.

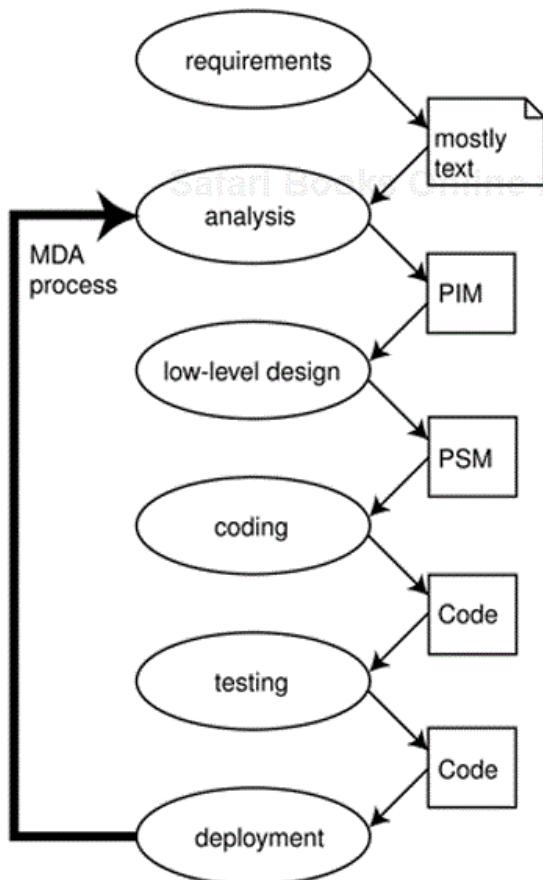


Figura 5.14: El ciclo de vida software bajo el enfoque **MDA** [Fuente:³⁰]

De una fase a la otra del ciclo de vida **MDA**, se pasa de forma automática mediante una transformación **M2M** (de **PIM** a **PSM**, para pasar del análisis al diseño) o una transformación **M2T** (generación de código), tal y como se muestra en la Figura 5.15.

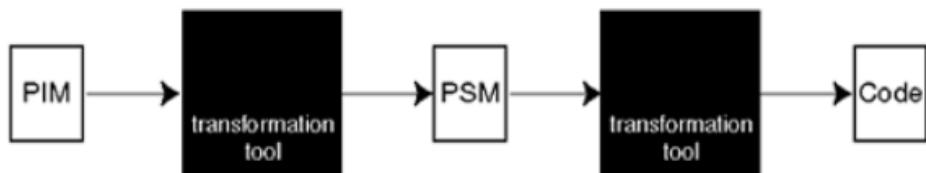


Figura 5.15: Los tres pasos más importantes en el proceso de desarrollo **MDA** [Fuente:³¹]

Los beneficios de MDA

Productividad Al automatizarse el paso del **PIM** al **PSM** (y de éste a código), los desarrolladores dedicarán todo su esfuerzo a la definición del **PIM**. Es cierto que definir la transformación de **PIM** a **PSM** es una tarea muy difícil y de alta especialización, pero una vez hecha, valdrá para usarla en el desarrollo de cualquier otro producto software. De esta forma, la productividad se aumenta por dos razones:

- Los desarrolladores del producto software tienen menos trabajo que hacer, pues solo especifican el **PIM** y se olvidan de los detalles para adaptar su modelo a las plataformas específicas donde se ejecutará el producto software. También tendrán que hacer algún trabajo manual en el **PSM** y en el código, pero será mucho menor
- Como consecuencia de lo anterior, pueden dedicar mucho más tiempo al **PIM**, y llegar a entender mucho mejor el problema que se quiere resolver, de forma que se conseguirá satisfacer mucho mejor desde el principio las necesidades de sus clientes y de los usuarios finales

No se puede olvidar que para que esto sea real, y no solo teórico, además de que existan herramientas capaces de generar de forma completamente automática el **PSM** a partir del **PIM**, también habrá que ser mucho más preciso al especificar el **PIM**. No hay que olvidar que un humano que lea el modelo en papel puede perdonar, pero una herramienta para transformación automática no puede³².

Portabilidad Al ser el **PIM**, por definición, independiente de la plataforma, y al poder transformarse automáticamente en múltiples **PSM** para diferentes plataformas, el enfoque **MDA** hace su especificaciones de alto nivel (**PIMs**) completamente portátiles siempre que haya herramientas disponibles para hacer las transformaciones.

Para las plataformas populares se puede esperar que vaya habiendo una gran cantidad de herramientas disponibles. Para plataformas menos populares, una solución es que el desarrollador defina transformaciones específicas para esa plataforma y que luego use una herramienta que permita incorporar nuevas transformaciones.

Cada vez que aparezca una nueva tecnología o una nueva plataforma, la industria software debe ser capaz de entregar las transformaciones correspondientes a tiempo. Esto nos permite implementar rápidamente nuevos sistemas con la nueva tecnología, a partir de los **PIM** que ya teníamos.

Interoperabilidad Las relaciones entre los distintos **PSMs** generados a partir de un **PIM** se llaman puentes, en el argot MLA (ver Figura 5.16). Los puentes transforman los conceptos de una plataforma en conceptos utilizados en otra plataforma. **MDA** garantiza la interoperabilidad porque, no solo genera los **PSMs**, sino los puentes que conectan los **PSMs** de distintas plataformas. Con un puente **PSM-PSM** podemos deducir cómo los elementos de un **PSM** se relacionan con los elementos del otro **PSM**. Como se ve en la figura, esto

³²Nota añadida: Esto me recuerda a lo que también ocurre con el resto de la naturaleza, la cual, por no tener conciencia a diferencia del ser humano, no puede perdonar los errores o agresiones contra ella.

significa generar también de forma automática un puente entre los códigos generados para ambas plataformas.

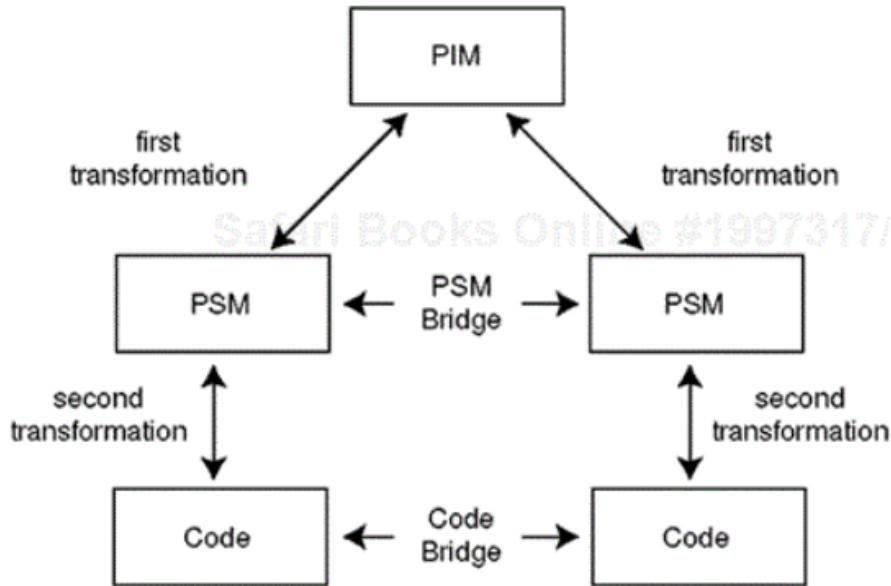


Figura 5.16: Interoperabilidad MDA usando puentes. [Fuente:³³]

Dado que conocemos todos los detalles técnicos específicos de las plataformas de la que tenemos PSMs (de lo contrario no podríamos haber realizado las transformaciones de **PIM** a los distintos PSMs que luego se traducirán a código), tenemos toda la información que necesitamos para generar un puente entre dos de esos PSMs capaz de conectar mediante otro puente los códigos generados por ambos PSMs.

Tomemos, por ejemplo, un **PSM** como modelo de un código Java y el otro **PSM** como modelo de una **BD** relacional. Dado un elemento *Cliente* en el **PIM**, sabemos en qué clase(s) de Java y en qué tabla(s) se traduce. Construir un puente entre un objeto Java en **Java-PSM** y una tabla en **BD-PSM** es fácil. Para recuperar un objeto de la **BD**, consultamos la(s) tabla(s) generada(s) desde *Cliente* e instanciamos la(s) clase(s) en el otro **PSM** con los datos consultados. Para almacenar un objeto, obtenemos los datos en el objeto Java y los almacenamos en la(s) tabla(s) “*Cliente*”.

La interoperabilidad multiplataforma se puede lograr mediante herramientas que no solo generan PSMs, sino también los puentes entre ellos, y posiblemente también a otras plataformas. Gracias a esta propiedad, los cambios tecnológicos podrán ser incorporados al producto software sin que afecten al **PIM** y por tanto a toda la inversión realizada para generararlo.

Mantenimiento y documentación El **PIM** cumple la función de documentación de alto nivel que se necesita para cualquier sistema software.

La gran diferencia es que el **PIM** no se abandona después de escribirlo. Los cambios realizados en el sistema eventualmente se realizarán cambiando el **PIM** y regenerando el

PSM y el código. En la actualidad, muchos de los cambios se realizan en el **PSM** y el código se regenera a partir de ahí. Sin embargo, se espera que puedan haber herramientas capaces de generar código desde el **PIM**, e incluso actualizarlo si se hicieran cambios en el **PSM**, de forma que la documentación de alto nivel siga siendo coherente con el código real.

Por otro lado, el enfoque **MDA** debe permitir anotar información adicional, que no se puede capturar en un **PIM**, como por ejemplo, la argumentación de las distintas decisiones tomadas durante el desarrollo del **PIM**.

El marco de referencia de MDA

La Figura 5.17 muestra los componentes básicos de **MDA**. Hay que señalar que, aunque los modelos **PIM** y **PSM** parezcan completamente diferentes, en la realidad se solapan más de lo que nos gustaría. Por ejemplo, un diagrama de clases en un **PIM** puede usar interfaces, y eso puede significar que está pensado para Java. O bien, un modelo en el que algunos componentes provengan de un antiguo software (legacy software) podría sugerir que el producto requerirá una cierta plataforma relacionada con el software antiguo. Lo que sí se puede decir es que el **PIM** debe ser más independiente de la plataforma y el **PSM** más dependiente de una plataforma, aunque la línea divisoria entre ambos no esté tan clara. La transformación **MDA** transformará **PIM** en **PSM**, aún siendo “independiente” y “específico” términos relativos.

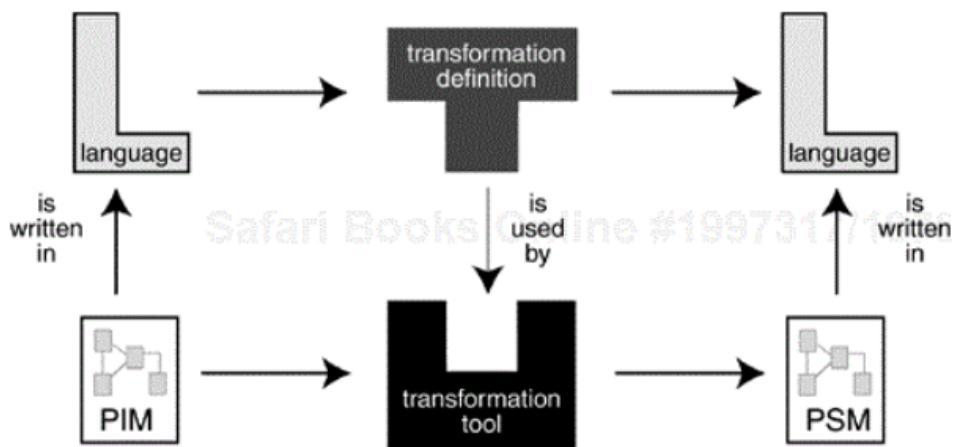
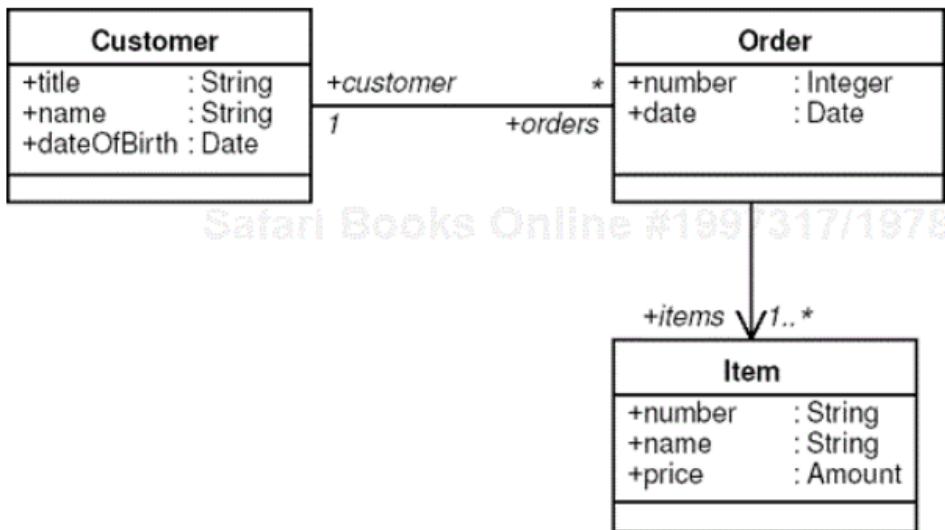
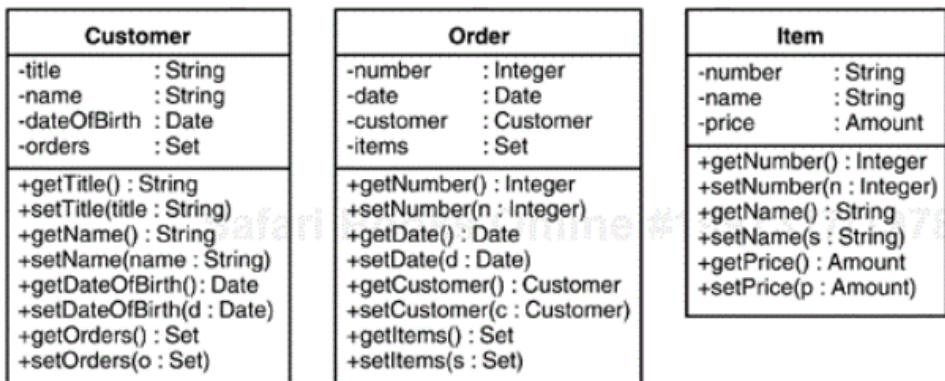


Figura 5.17: Marco de referencia **MDA**. Visión de conjunto. [Fuente:^{34]}

Ejemplo: transformación entre dos modelos UML

Partiendo de un **PIM** (Figura 5.18), que es un diagrama de clases en UML, vamos a transformarlo en un **PSM** (Figura 5.19) donde la plataforma específica será Java, y viceversa. Veremos así qué reglas se usan para entender cómo el proceso podría ser automatizado.

Figura 5.18: Platform Independent Model. [Fuente:³⁵]Figura 5.19: Platform Specific Model targeted at Java. [Fuente:³⁶]

En el **PIM** se modelan conceptos que describen el qué, relacionados directamente con la operatividad empresarial. Las clases se corresponden con conceptos relacionados con la parte del objeto social de la empresa que se informatizará, y por tanto, con conceptos en la parte del modelo de negocios o **CIM** que se va a informatizar. Los atributos de instancia significan propiedades cambiables que tienen las distintas ocurrencias de un concepto del ámbito de la empresa sobre el que se quiere hacer un sistema software. Por ello se ponen en el **PIM** como atributos (es decir variables asociadas a una ocurrencia de un concepto), y públicas (es decir, permiten mostrar dicha propiedad).

De atributos públicos a privados

En el **PSM** ya no modelamos conceptos de la actividad u objeto social de la empresa, sino cómo eso se codificará en un sistema computacional concreto (plataforma). Para ello,

no puede olvidarse los principios de ocultamiento y encapsulamiento propios de Java u otros lenguajes OO. Por estos principios, el concepto es una clase de forma que cada objeto realiza las tareas relacionadas con él y controla el acceso a sus atributos. Así, los atributos son privados y se accede a ellos por operaciones bien definidas (consultores y modificadores básicos).

Se pueden derivar por tanto las siguientes reglas de transformación:

- Se mantiene el nombre de las clases: Para cada clase *className* en el **PIM**, hay una clase *className* en el **PSM**
- Se privatizan los atributos y se definen métodos consultores y modificadores básicos para todos los atributos: Para cada atributo público *attributeName: Type* de una clase *className* en el **PIM**, se definen los siguientes atributos y métodos en la clase *className* del **PSM**:
 - Un atributo privado de igual nombre y tipo: *attributeName: Type*
 - Un método público con igual nombre que el atributo precedido por *get* y que devuelva el tipo del atributo: *getAttributeName(): Type*
 - Un método público con igual nombre que el atributo precedido por *set*, con el atributo como parámetro y sin valor de devolución: *setAttributeName(att : Type)*

Las transformación inversa, de **PSM** a **PIM** tendrá las siguientes reglas:

- Se mantiene el nombre de las clases: Para cada clase *className* del **PSM** hay una clase *className* en el **PIM**
- Se quitan los métodos de acceso básicos (consultores y modificadores básicos) y sus atributos correspondientes se hacen públicos: Para cada combinación en el **PSM** de los siguientes atributos y métodos en el **PSM**:
 - Un atributo privado *attributeName: Type*
 - Un método público *getAttributeName(): Type*
 - Un método público *setAttributeName(att : Type)*existirá un atributo público *attributeName: Type* en la clase correspondiente *className* del **PIM**.

A partir de aquí, también se pueden definir transformaciones del **PSM** a código Java y viceversa.

Asociaciones

Para transformar las asociaciones del **PIM** al **PSM** para una plataforma Java, usamos las siguientes reglas de transformación:

- Para cada asociación no dirigida (doble navegabilidad) en el **PIM**, en el **PSM** tendremos:
 - Para cada extremo de la asociación con nombre de rol *role* en el **PIM**, hay un atributo de igual nombre en la clase opuesta del **PSM**
 - El tipo de este atributo es la clase en ese extremo de la asociación si la multiplicidad es 0 o 1, y Set si la multiplicidad es mayor
 - El atributo creado en el **PSM** será privado y tendrá definidos su consultor y modificador básicos (*get* y *set*) como el resto de atributos
 - Para cada asociación dirigida en el **PIM**, en el **PSM** tendremos lo especificado para las asociaciones no dirigidas pero solo en el extremo final de la asociación (en el sentido de la navegabilidad)

Hay algunas decisiones de diseño que son aplicadas por las distintas reglas de transformación y que no podemos cambiar. Sin embargo existe una posibilidad de que los desarrolladores de un sistema concreto puedan cambiar una transformación por defecto, es lo que se llama ajustar (del inglés, tuning) una transformación.

Estándares **OMG** para MDA

Los siguientes estándares **OMG** son usados por **MDA**:

MOF: Meta Object Facility

Es el componente central de su arquitectura de metamodelado.³⁷ Puede ser usado para que otras empresas desarrolle sus propios estándares **MD**. **MOF**, como se vio en el apartado ??, es un lenguaje especial que permite definir cualquier lenguaje que se quiera usar, de forma que se garantiza que está definido formalmente y se podrá usar el enfoque **MDA** para definir transformaciones entre estos lenguajes y cualquier lenguaje estándar **OMG** (los lenguajes estándares de **OMG** garantizan siempre su definición formal).

QVT: Query/View/Transformation

Se trata de un conjunto estándar de lenguajes tanto para consultas y vistas sobre el modelo como para transformaciones.³⁸

Es un estándar de **OMG**. Las consultas y las vistas se consideran como casos especiales de las transformaciones.

Lenguajes **OMG**

Además de UML, el lenguaje de modelado más ampliamente usados, veremos algunos otros.

³⁷Rodrigues da Silva, «Model-Driven Engineering: A survey supported by a unified conceptual model».

³⁸Rodrigues da Silva.

OCL

OCL (del inglés, Object Constraint Language), es un lenguaje de consulta y expresiones, que forma parte del estándar UML. El término “restricción” viene de que en sus primeras versiones solo se usaba para especificar restricciones sobre los modelos UML. En la actualidad es un lenguaje de consultas completo, con el mismo poder expresivo que pueda tener SQL.

Profiles UML

Se trata de extender UML con definición de estereotipos, etiquetas y restricciones (usando **OCL**), como una forma simple pero práctica de definir lenguajes de modelado específicos (**D(S)ML**) gráficos, a partir de UML. Sin embargo, se les objeta que se puedan añadir propiedades al elemento original pero no eliminarlas o inhibirlas. Además, las herramientas de modelado que admiten perfiles UML generalmente no verifican adecuadamente la calidad de estos modelos. Su mayor ventaja es su compatibilidad con herramientas CASE UML y entornos de desarrollo, lo que hace que se usen también con bastante frecuencia para definir DS(M)Ls.³⁹

Los perfiles UML se utilizan con frecuencia para definir PSMs.

Algunos perfiles UML que son estándares de **OMG** son:

- **CORBA** (del inglés, Common Object Request Broker Architecture)
- **EDOC**, del inglés Enterprise Distributed Object Computing
- **EAI**, del inglés Enterprise Application Integration
- **Scheduling, Performance, and Time**

Action Semantics (AS)

Es un lenguaje que permite definir la semántica de los modelos conductuales de UML. El mayor problema es que definen la conducta a muy bajo nivel, de forma que no puede usarse para describir PIMs. Además, por definir solo la semántica, pero no una sintaxis concreta, no puede ser usado directamente de una forma estándar.

Lenguajes para PIM

Lo más importante para sacar partido real de **MDA** es tener definidas transformaciones de **PIM** a **PSM**. Para ello necesitamos un lenguaje formal, con un altísimo nivel de completitud, consistencia y precisión (ausencia de ambigüedad) para definir los PIMs.

Veremos aquí las posibilidades de UML y sus extensiones para poder especificar un **PIM**.

³⁹Rodrigues da Silva, «Model-Driven Engineering: A survey supported by a unified conceptual model».

UML básico como lenguaje PIM

El punto fuerte de UML es el modelado estructural de un sistema, lo que se hace mediante el modelo de clases y que permite generar el **PSM** de forma automática, como se ha visto en el ejemplo.

Pero presenta otros problemas que impiden que se puedan automatizar la generación de un **PSM** a partir de un **PIM**. En concreto, para modelar la parte dinámica o conductual de un sistema, UML ofrece diversos diagramas pero todos adolecen de falta de completitud y formalismo, piénsese por ejemplo en un diagrama de casos de uso o en un diagrama de interacción independiente de la plataforma.

Si se genera un **PSM** a partir de un **PIM** especificado con UML, quedarán muchos aspectos que habrá que hacer a mano, relacionados con la parte dinámica del sistema.

UML ejecutable

UML ejecutable⁴⁰ se define como una combinación entre UML básico y **AS**. Su sintaxis concreta no está estandarizada.

Por partir de UML, presenta todas las ventajas de UML para realizar modelos **PIM** de la estructura de un sistema software. Pero además, al añadir **AS**, se mejora la capacidad para modelar la conducta del sistema. Las conductas se definen usando especialmente las máquinas de estado, de forma que además a cada estado se le asocia un procedimiento escrito en **AS**.

Aunque teóricamente es capaz de especificar un **PIM** y generar a partir de él un **PSM** completo, presenta algunos problemas:

- No todos los dominios permiten el uso adecuado de máquinas de estado como modelos para especificar su dinámica
- **AS** no es un lenguaje de muy alto nivel, sino que usa conceptos más bien al nivel del **PSM**, i.e., específicos de la plataforma. Por tanto, serán pocos los beneficios a nivel de especificación de la conducta del sistema que puedan obtenerse de este enfoque comparados con escribir directamente el **PSM**
- El lenguaje **AS** no tiene una sintaxis o notación concreta estandarizada

Combinado UML-OCL

Con esta combinación se pueden conseguir PIMs de alta calidad: consistentes, precisos y con toda la información. El aspecto estructural fuerte de UML se complementa con el uso de expresiones **OCL** para definir el cuerpo de las operaciones de consulta del sistema y reglas de la operatividad del negocio, incluyendo disparadores dinámicos.

Con pre y post-condiciones se puede expresar la dinámica del sistema, pues el cuerpo de la operación podría ser generado a partir de la post-condición. Sin embargo, la mayoría de las veces esto no puede hacerse de forma automática y hay que escribirlo a mano en el **PSM**.

⁴⁰Stephen J. Mellor y Marc J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture* (USA: Addison-Wesley, 2002).

5.3. Modelado de procesos de negocio

Los conceptos de “modelado de procesos de negocio” (**BPM**, del inglés “business process modeling”) y de “modelado de negocio” (**BM**, del inglés “business modeling”), suelen usarse de forma intercambiable, aunque en otros contextos tienen significados distintos. Nosotros aquí nos referimos a los términos sinónimos que se refieren a la abstracción que se hace de las actividades realizadas en una empresa para llevar a cabo las tareas que constituyen su negocio. Antes se le solía referir como “flujo de trabajo” (“workflow”). También se suele usar como sinónimo el término “gestión de procesos de negocio” (también con las siglas **BPM**, del inglés “business process management”).

Aun así, veamos los diferentes y grandes matices del concepto:

“Se trata de modelar un proceso empresarial, utilizando representaciones gráficas y XML estándar, como un flujo de actividades. Gracias a algunos estándares (especialmente **BPEL** y **BPMN**, que se verán más adelante), la semántica de un flujo está tan rigurosamente definida como la de cualquier lenguaje de programación: cualquiera puede bosquejar informalmente cuadros y flechas en una pizarra durante una reunión de análisis de requisitos, pero un proceso **BPM** está diseñado con la ejecución en mente.

BPM también está muy relacionado con la noción de “Arquitectura Orientada a Servicios” (**SOA**, del inglés “Service Oriented Architecture”⁴¹). Mientras que el uso tradicional de un flujo de trabajo se refería al movimiento de trabajo de persona a persona dentro de una organización, los procesos **BPM** contemporáneos están diseñados para interactuar como servicios con otros sistemas, o incluso para coordinar otros sistemas, incluidos los procesos comerciales de otras compañías. De hecho, un proceso de negocio es un servicio, destinado a ser llamado por otros sistemas, y estas llamadas disparan su ejecución. Darse cuenta de este hecho es uno de los primeros grandes pasos para comprender **BPM**.⁴²

“Es una materia interdisciplinaria que se ocupa del desarrollo continuo de los procesos de negocio con la ayuda de la tecnología de la información (**TI**). **BPM** ha ganado mucha popularidad en los últimos años debido al auge de la arquitectura orientada a servicios (**SOA**) y tecnologías relacionadas que se pueden aplicar para ejecutar y administrar procesos comerciales basados en **TIs**. Para conseguir que los requisitos del negocio se puedan traducir a procesos ejecutable, el modelado debe comenzar por los procesos de más alto nivel, siendo luego refinados por modelos más detallados y exactos hasta producir finalmente código ejecutable. Este desarrollo se describe en la Figura 5.20. Para mejorar la productividad de la organización mediante las **TIs** y la mantenibilidad de los modelos, las transformaciones entre modelos y código deben automatizarse tanto como sea

⁴¹Nota añadida: Se trata de un estilo o patrón arquitectónico, a menudo es utilizado para definir servicios Web.

⁴²Michael Havey, *Essential Business Process Modeling* (USA: O'Reilly Media, Inc., 2005).

posible. Este tipo de enfoque puede relacionarse con MDA, aunque aquí no se exige el uso de las tecnologías relacionadas con MDA.⁴³

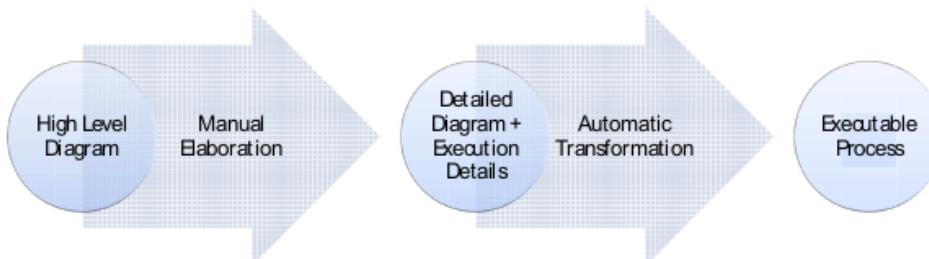


Figura 5.20: Desarrollo desde un diagrama del proceso de negocio hasta procesos ejecutables. [Fuente:⁴⁴]

Mika Koskela y Jyrki Haajanen, del VTT (Technical Research Center, Finlandia), redactaron en 2007 un informe (“Business Process Modeling and Execution”⁴⁵) para evaluar hasta qué punto las tecnologías BPM de última generación cumplen la promesa de que el código ejecutable pueda generarse automáticamente a partir de modelos de procesos de negocio. A partir de aquí incluiremos una traducción literal, aunque resumida, de este informe. Para evitar continuas citas, se incluirán solo las de trabajos adicionales.

En los últimos años se está desarrollando mucho la automatización e integración de procesos de negocios, en gran parte debido a la habilitación de estándares y tecnologías relacionados con SOA. Sin embargo, dominar BPM no es una tarea sencilla. Para muchas empresas se hace difícil por la gran variedad de opciones disponibles y promesas y afirmaciones contradictorias que se realizan. Por ello, es importante conocer qué se puede esperar de las tecnologías BPM en la actualidad, y saber cómo orientar posibles futuras inversiones en TIs relacionadas con BPM.

Estudiaremos la relación entre los lenguajes de modelado y los lenguajes de ejecución, es decir, hasta qué punto es posible transformar los modelos de procesos de negocio a procesos ejecutables y, en cierta medida, viceversa. A menudo las herramientas software o los estándares no clarifican bien esto, por eso es interesante conocerlo.

5.3.1. Modelado y ejecución de procesos de negocio

Lenguajes de modelado

El modelado visual de procesos de negocio, es decir, dibujar un diagrama de proceso de negocio, es en teoría el primer paso del ciclo de vida del proceso de negocio. Existen numerosas anotaciones para este propósito. En el nivel más alto de abstracción, podemos incluso visualizar los elementos más importantes usando formas y líneas arbitrarias. Los lenguajes basados en XML a veces también son categorizados como lenguajes de modelado. Pero nos

⁴³Mika Koskela y Jyrki Haajanen, *Business Process Modeling and Execution. Research Notes 2407*, informe técnico (2007), <https://www.vttresearch.com/sites/default/files/pdf/tiedotteet/2007/T2407.pdf>.

⁴⁵Koskela y Haajanen.

vamos a centrar solo en lenguajes **BPM** visuales que estén suficientemente detallados como para expresar procesos ejecutables, de forma que el código pueda generarse en función de los diagramas. Usamos el término “lenguaje de modelado” (**ML**), del inglés “Modeling Language”) en lugar del de “notación de modelado” para enfatizar que los estándares que vamos a ver, pretenden especificar el comportamiento detallado que indican los elementos de modelado y no solo la notación visual.

Si quitamos lenguajes propietarios, lenguajes con fines académicos, lenguajes sin notación visual y lenguajes que no pretendan modelar procesos ejecutables, nos quedamos solo con dos lenguajes que merezcan la pena:

- **BPMN** (de inglés, Business Process Modeling Notation), de **OMG** 2006
- **AD** (del inglés, Activity Diagram), un diagrama UML, de **OMG** 2005

Las diferencias entre ellos, según los criterios de expresividad y de soporte, son:

- Expresividad (capacidad de representar distintas clases de constructos de processos, patrones y situaciones propias de los procesos de negocio).- Realmente no hay grandes diferencias, aunque quizás **BPMN** puede ser más potente a veces en cuanto a capacidad de expresión. En general, para ambos lenguajes suele ocurrir que a veces la especificación que permiten sea demasiado imprecisa, sin quedar claro si pueden representar ciertos patrones
- Soporte.- Es el factor decisivo. Hay muy pocas herramientas para **BPM** basadas en **Activity Diagrama (AD)**s, lo que hace a **BPMN** más accesible como lenguaje para **BPM**, tanto para los usuarios TIs (informáticos) como para los usuarios del negocio (gestores)

BPMN, un ejemplo de lenguaje de modelado

BPMN es uno de los lenguajes de modelado de procesos empresariales más recientes disponibles, pero ya se ha hecho bastante popular: más de 40 implementaciones.

Por un lado, proporciona una notación que tanto usuarios comerciales y de TIs puedan entender. Por otro lado, está diseñado para actuar como una notación visual para lenguajes ejecutables. Puede representar procesos privados, públicos y de colaboración. La idea es simplemente que uno puede elegir libremente el nivel de granularidad que necesita al modelar procesos concretos. La mayoría de los elementos **BPMN** pueden ser traducidos hasta poder llevarlos a ejecución, pero algunos se utilizan únicamente con fines informativos. Una de las deficiencias de **BPMN** es que carece de una semántica formal y que las especificaciones para ciertos elementos tampoco son adecuadas para fines de ejecución. Además, la especificación no incluye un formato de intercambio XML para diagramas **BPMN**. **OMG** lo ha hecho después con la especificación del “Metamodelo de Definición del Proceso de Negocio” (**BPDM**, del inglés “Business Process Definition Metamodel”), pero aún no es compatible con las herramientas disponibles.

Conceptos básicos de BPMN

Aprender BPMN no es excesivamente complicado, especialmente si se está familiarizado con algún lenguaje BPM. Aquí solo se presenta una idea general.

En BPMN, un “Diagrama de Proceso de Negocio” (BPD, del inglés “Business Process Diagram”) es una presentación visual de un proceso de negocio. La especificación no hace una distinción clara entre este concepto y el de *modelo*, sin embargo, el término *modelo* generalmente implica que también se han especificado los detalles de ejecución necesarios (por ejemplo, información variable). Un BPD consiste en elementos cuya apariencia visual y semántica se definen en la especificación BPMN. Sus elementos son las *Activities*, y los actores que las llevan a cabo. Los actores están representados por *Pools* (grupos), que a su vez pueden ser subdivididos en *Lanes*. Esto se utiliza exclusivamente con fines informativos, sin afectar para nada a la ejecución del proceso. Además existen otros elementos, que se describen a continuación.

El progreso del proceso está determinado por los *Events*. Dentro de un grupo, que a menudo representa un sistema de TI independiente, la secuencia de actividades está representada por *SequenceFlows*. La interacción con otros grupos se indica mediante *MessageFlows*. El progreso del flujo de secuencia a menudo se describe utilizando el concepto de *token*, que pasa a través de los objetos del flujo. Cuando se discute sobre el flujo de control del proceso, se usa el término *upstream* para referirse a los elementos o tokens que aparecen o se generan antes del punto discutido en el proceso y el término *downstream* para referirse a las partes que siguen al punto en discusión. El flujo de control puede divergir según las condiciones especificadas. Las condiciones, o bien están directamente vinculadas al flujo, o se usan distintos tipos de *Gateways* (pasarelas o puertas de enlace) para indicar puntos de decisión. La Figura 5.21 muestra estos elementos básicos.

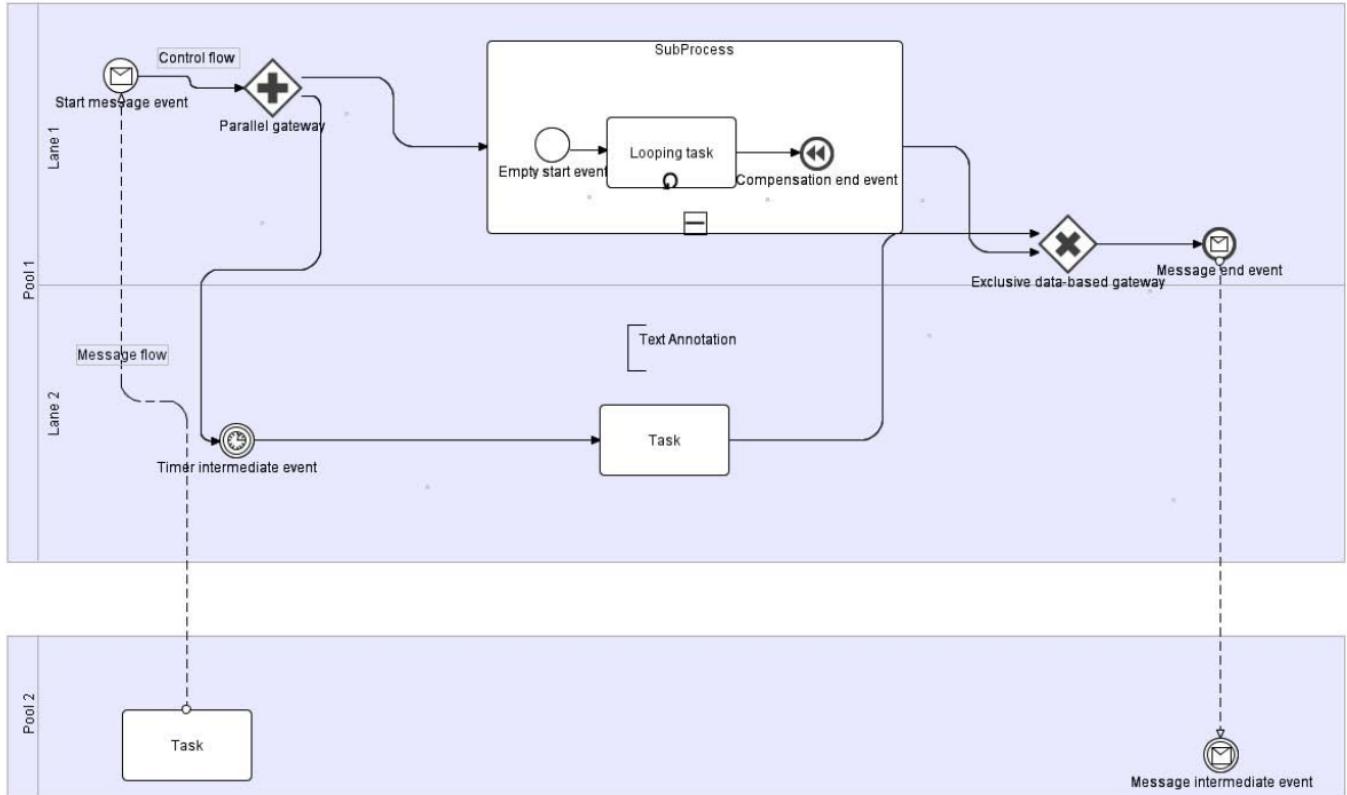


Figura 5.21: Ejemplos de elementos BPMN. [Fuente:⁴⁶]

Semántica de los elementos

BPMN define varias modalidades o subtipos de cada uno de los distintos elementos. Por ejemplo, la *Activity* puede ser un subprocesso o una actividad de tipo bucle. Un *Event* puede ser a su vez *Start* (indican el comienzo del proceso), *Intermediate* (indican un estado intermedio del proceso) o *Final* (indican el fin del proceso), y puede representar un mensaje entrante o saliente o un error. Dentro de los elementos finales, hay varios tipos según los diferentes resultados del proceso que definan. Un evento final muy típico es el *MessageEndEvent*, que implica que se envía un mensaje como resultado del proceso. Otros tipos incluyen, por ejemplo, *Error* (se lanza un error), *Compensation* (el proceso vuelve hacia atrás), *Terminate* (todas las actividades en el proceso finalizan inmediatamente) y *Empty* (el resultado no está especificado). Todos los eventos finales, salvo los de terminar, deben consumir todos los tokens del proceso para que la instancia del proceso pueda terminar.

Además de las actividades y sus conectores, BPMN define *Artifacts*, como *DataObjects* que pueden ser anotados para indicar las entradas y salidas de las actividades.

Las *Gateways* se utilizan para controlar el flujo de secuencia y, en situaciones complejas, son de uso obligatorio. Hay dos tipos básicos de puertas de enlace: *Databased*, que evalúa la ruta a elegir en función de, por ejemplo, valores de variables y *Eventbased*, que la elige, por ejemplo, en función de un mensaje que le llega. Las puertas pueden dividirse (split),

bifurcarse (fork) o fusionarse (merge), es decir, pueden dividir un flujo de secuencia en varios flujos alternativos (división) o en varios flujos paralelos (bifurcación) o combinar flujos (fusión). La bifurcación o la fusión sin puertas de enlace se llama **incontrolada**. Otro criterio de clasificación divide a las puertas de enlace en:

- Inclusivas (OR).- se pueden tomar todas las combinaciones posibles de rutas independientes. En procesos paralelos (división), se definen múltiples rutas que se ejecutan simultáneamente. En la bifurcación a menudo se puede dejar algunos. Al fusionarse, indican que todas las entradas deben estar disponibles antes de continuar la ejecución.
- Exclusivas (XOR).- el flujo de secuencia solo puede tomar uno de los caminos salientes. En la fusión, a menudo no se requieren. En procesos paralelos (división) se crean múltiples tokens que se combinan nuevamente en la fase de fusión.
- Paralelas o complejas (AND).- Se pueden utilizar para combinar el comportamiento de varias puertas de enlace vinculadas. El comportamiento de bifurcación o fusión se puede determinar mediante una expresión que puede, por ejemplo, evaluar los datos del proceso para determinar qué flujos seleccionar.

Las Tablas 5.1 y 5.2 explican de modo informal la semántica en BPMN de diferentes divisiones (tabla 5.1) y uniones (tabla 5.1). Se muestran diagramas de ejemplo en las Figuras 5.22 y 5.23. Cabe señalar que la unión A es solo un caso especial de la unión B (Figura 5.23), pero se presenta por separado para aclarar el comportamiento indicado por una fusión no sincronizada en BPMN.

Detalle	Descripción
Constructo	Explicación
División: división-XOR (XOR)	El flujo de control se divide en más de una rama Solo se puede elegir una durante la ejecución.
División: división-OR (OR)	Un punto de ramificación de caminos independientes: Se puede tomar cualquier combinación de rutas.
Bifurcación: división-AND (división paralela)	Un punto de ramificación donde un solo flujo se divide en dos o más caminos paralelos. Se envía un token a cada uno de los flujos salientes.

Tabla 5.1: Comportamiento de los procesos en divisiones BPMN [Fuente.⁴⁷

Cabe señalar que en muchos casos se puede realizar exactamente el mismo comportamiento tanto por pasarelas como por flujo no controlado. Un ejemplo de esto se da en el contexto de la división-AND de la Figura 5.22. En la división OR-XOR, se debe usar un

⁴⁷Mika Koskela y Jyrki Haajanen, *Business Process Modeling and Execution. Research Notes 2407*, informe técnico (2007), Tabla 1, <https://www.vttresearch.com/sites/default/files/pdf/tiedotteet/2007/T2407.pdf>.

⁴⁸Mika Koskela y Jyrki Haajanen, *Business Process Modeling and Execution. Research Notes 2407*, informe técnico (2007), Tabla 2, <https://www.vttresearch.com/sites/default/files/pdf/tiedotteet/2007/T2407.pdf>.

Detalle	Descripción
Constructo	Explicación
Unión: unión-XOR	Se combinan varios flujos en uno. El proceso continúa en cuanto alguno de los tokens alcance ese punto.
Unión: unión-XOR: de actividades	El proceso continúa en cuanto algún flujo entrante produzca un token. Cada vez que se produzca un nuevo token se creará una nueva instancia de la actividad de unión.
Unión: unión-OR (división paralela)	Un punto de ramificación donde un solo flujo se divide en dos o más caminos paralelos. Se envía un token a cada uno de los flujos salientes.
Unión: unión-AND	Se sincronizan múltiples caminos de flujo. El proceso espera a que todos los flujos entrantes produzcan un token para continuar

Tabla 5.2: Comportamiento de los procesos en uniones BPMN [Fuente:⁴⁸

flujo de secuencia condicional (flechas con rombos) si se omiten las puertas de enlace. Las condiciones adjuntas indican si la elección es o no exclusiva (obsérvese, por ejemplo, que en la división-XOR en la Figura 5.22, es implícito que se puede elegir sí o no, pero no ambos). Las puertas de enlace indican explícitamente que las rutas alternativas son exclusivas, mientras que si se usa flujo no controlado, esta restricción debe constar en las condiciones adjuntas al flujo de control. También se debe tener en cuenta que la puerta de enlace exclusivas (XOR) pueden ser modeladas con o sin el dibujo: el comportamiento del proceso indicado es exactamente el mismo.

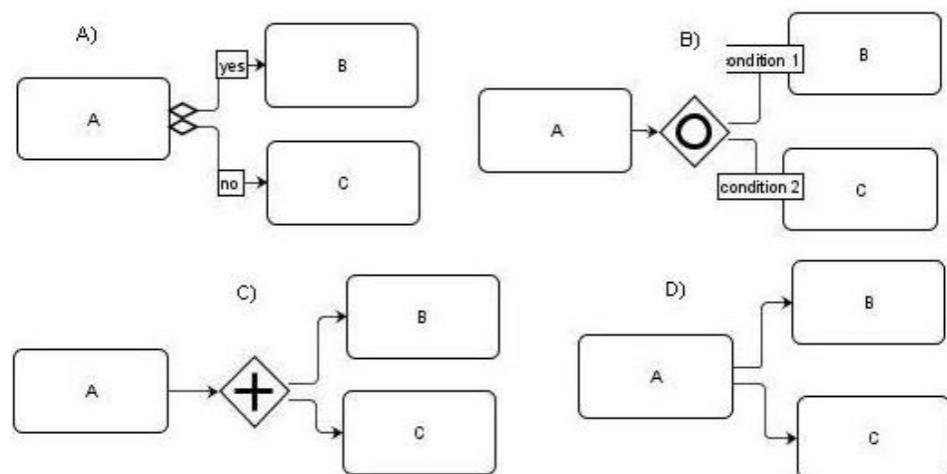


Figura 5.22: Divisiones BPMN: A) XOR; B) OR; C) y D): AND (comportamientos equivalentes). [Fuente:⁴⁹]

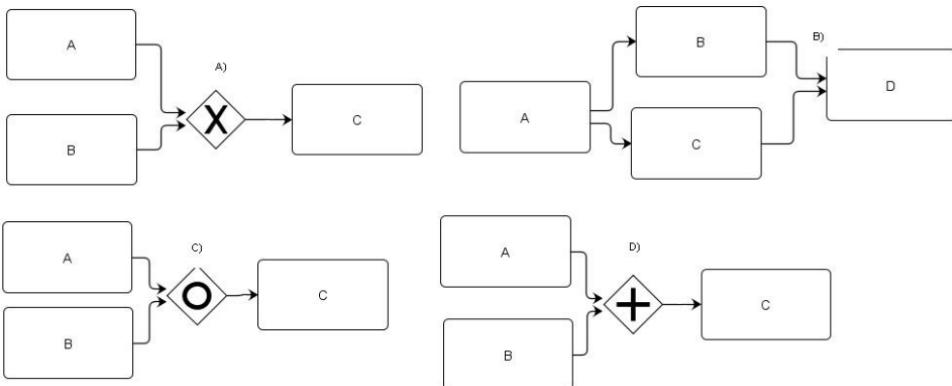


Figura 5.23: Uniones **BPMN**: A) XOR; B) XOR (actividades paralelas); C) OR; D) AND.
[Fuente:⁵⁰]

Inmadurez del estándar

Hay algunos problemas en la notación que muestran que el estándar **BPMN** aún no está maduro.

Por ejemplo, según estas tablas, se puede decir que expresar una división del proceso en **BPMN** es muy claro Y fácil de entender. Sin embargo, expresar la unión no es tan sencillo y es probable que se creen malentendidos entre personas familiarizadas con otras anotaciones de modelado. Por ejemplo, no es muy obvio que el flujo de secuencia que se fusiona sin puertas de enlace proceda de caminos independientes, creando así múltiples instancias de su actividad común si los flujos son paralelos. En los **ADs** de UML, por ejemplo, se usa un nodo de unión para expresar este tipo de situaciones. Por otro lado, según el documento de especificación **BPMN**, no queda totalmente claro cómo se determina el número posible de tokens entrantes. La unión OR también se ha considerado mal especificada y por ello no se usa en los casos de prueba.

Además, debido a que la semántica formal no está incluida en la especificación, es posible que los vendedores de herramientas malinterpretan las reglas. Esto da como resultado una expresión de comportamiento potencialmente incorrecta, aunque pudiera crearse un código ejecutable basado en el modelo.

Complejidad de la notación

Por otro lado, si el uso de divisiones y uniones no se analiza cuidadosamente, se pueden crear modelos erróneos, que pueden generar código ejecutable de forma precisa que de problemas de ejecución.

Se pueden identificar tres tipos de problemas:

- Puntos muertos (deadlocks).- La ejecución cesa porque el proceso espera tokens que nunca se generará. Esto puede suceder, por ejemplo, si las divisiones OR y AND no se emparejan bien.

- Bloqueos en vivo (livelocks).- El proceso cambia constantemente su estado, pero no progresa, por ejemplo, un bucle infinito.
- Múltiples instancias.- Si se instancia varias veces la misma actividad puede producirse un consumo innecesario de recursos, o incluso resultados incorrectos de la operativa del negocio.

Un ejemplo BPMN⁵¹

Havey⁵² pone un ejemplo básico (ejemplo “Hello World!”) para BPMN, que representa un proceso para la gestión de reclamaciones en una compañía de seguros (ver Figura 5.24).

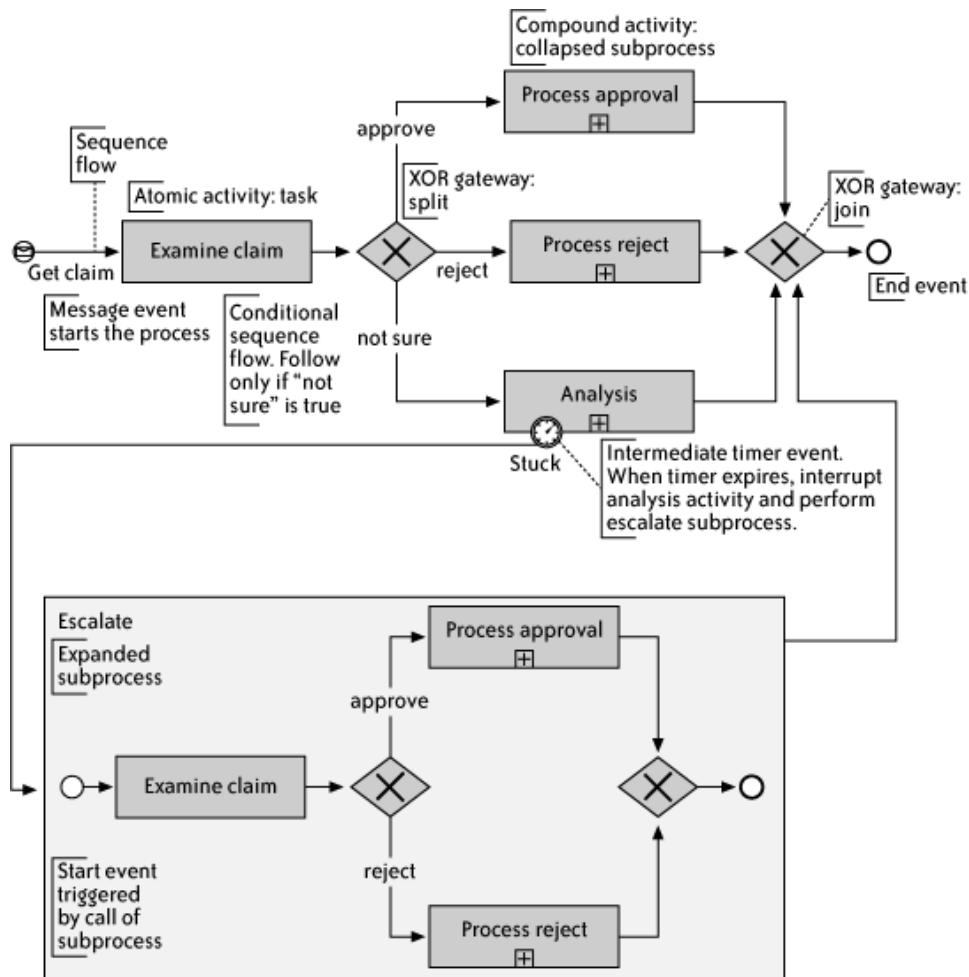


Figura 5.24: Proceso de reclamaciones de una empresa de seguros, usando BPMN. [Fuente: ⁵³]

El proceso recibe una reclamación (**Get claim**), la examina (**Examine claim**), y luego se divide en una de tres direcciones, dependiendo de si la reclamación ha sido aprobada (**Process approval**), rechazada (**Proceso reject**) o pendiente de un análisis posterior

⁵¹Havey, *Essential Business Process Modeling*.

⁵²Havey.

(**Analysis**). La opción de análisis tiene un límite de tiempo; si no se realiza con la suficiente rapidez, se cancela (**Stuck**) y se ejecuta un proceso especial de escalada (**Escalate**, cuyos pasos se incluyen en el cuadro **Escalar**). El proceso de escalado se comporta de manera similar a su padre: comienza examinando la reclamación, luego la aprueba o rechaza; no se permiten análisis adicionales en una escalada. El proceso padre se completa cuando se completa su ruta condicional (aprobación, rechazo, análisis o escalada).

En este diagrama se utilizan varios tipos de símbolos: eventos, puertas de enlace, actividades atómicas (o tareas), actividades compuestas (o subprocesos), flujo de secuencia y anotaciones de texto. Los eventos, dibujados como círculos pequeños, marcan el inicio (por ejemplo, **Get claim**) y los puntos finales del proceso, así como la condición de tiempo de espera intermedio (**Stuck**) que ocurre durante el análisis. Las puertas de enlace (rombos) ayudan a marcar la división condicional y unir parte del proceso. Las actividades (recuadros) representan el trabajo real realizado. Las tareas (por ejemplo, **Examine claim**) son acciones únicas, mientras que los subprocesos realizan una lógica arbitrariamente compleja. Un subproceso puede dibujarse:

- contraído (por ejemplo, **Process reject**): se dibuja con el signo +, y sus detalles quedan ocultos (se supone que están documentados en otro diagrama)
- expandido (por ejemplo, **Escalar**): tiene su lógica interna dibujada dentro de él. El flujo de secuencia es el conjunto de flechas que conectan las otras piezas; Las flechas etiquetadas con texto (p. ej., la flecha entre la puerta de enlace y **Process aproval**) son condicionales, seguidas solo si la condición es verdadera.

Las anotaciones de texto (cuadros abiertos) presentan comentarios instructivos.

Lenguajes ejecutables

Aunque en el campo de **BPM** existen varios lenguajes basados en XML, pocos de ellos se pueden utilizar directamente para ejecutar procesos comerciales automatizados. Estudiamos aquí **BPEL** (del inglés, Business Process Execution Language), por ser el único lenguaje que se consideró adecuado para la ejecución automatizada.

BPEL, un ejemplo de lenguaje ejecutable

BPEL es un lenguaje basado en XML para especificar procesos de negocios en el entorno de servicios Web. Se usa junto con **WSDL** (del inglés Web Services Description Languages) y otras tecnologías relacionadas. Esto significa que **BPEL** se usa para definir cómo se construye el proceso de negocio a partir de invocaciones a servicios Web y qué tipo de interacciones con participantes externos implica el proceso.

Como la mayoría de los lenguajes de programación, **BPEL** es complejo y la especificación es extensa. La comprensión profunda de **BPEL** requiere competencia en el desarrollo de software, así como conocimiento de las tecnologías de servicio Web en el que se basa. Aquí se hace solo una pequeña introducción.

El estándar **BPEL** actual es el resultado de la combinación de diferentes especificaciones desarrolladas por diferentes organizaciones, siendo las primeras Microsoft e IBM.

Principios de BPEL

BPEL se puede utilizar para especificar procesos ejecutables y abstractos. En el primer caso, se define la lógica de implementación completa del proceso, mientras que en el último caso solo se incluye el intercambio de mensajes entre los participantes del proceso. Los elementos centrales de un documento BPEL incluyen:

- **Roles** de los participantes del proceso
- **Tipos de puertos** requeridos por los participantes
- **Orquestación**: el flujo del proceso real
- **Información de correlación**: definición de cómo se pueden enrutar los mensajes para corregir instancias de composición, es decir, cómo asignar especificaciones abstractas a instancias en ejecución.

Aquí nos centramos en los aspectos ejecutables de BPEL.

Cuando se llevan a cabo transformaciones entre lenguajes de procesos de negocio, el paradigma de representación adoptado es una cuestión especialmente importante. Hay dos categorías básicas:

- Lenguajes orientados a bloques.- definen el flujo de control mediante el anidamiento de diferentes tipos de primitivas de control.
- Lenguajes orientados a gráficos: definen el flujo de control usando diferentes tipos de nodos y arcos. Los arcos conectan nodos entre sí, mediante conexiones temporales o lógicas.

Las transformaciones entre estos lenguajes, son difíciles, dado que los lenguajes orientados a gráficos puede expresar patrones de proceso que los lenguajes orientados a bloques no pueden. BPEL es en su mayor parte orientado a bloques. Las estructuras orientadas a gráficos pueden expresarse de forma parcial mediante el uso de enlaces. Por el contrario, los lenguajes de modelado son generalmente orientados a gráficos.

Elementos básicos de BPEL //

Las actividades de BPEL pueden ser actividades básicas o estructuradas. Las actividades básicas corresponden a componentes reales del proceso. Se realizan a través de interacción con el servicio Web, es decir, a través de invocaciones de operaciones WSDL. Estas actividades incluyen, por ejemplo:

- <invoke>: llama a una operación
- <receive>: espera un mensaje de entrada
- <reply>: envía un mensaje de salida
- <assign>: actualiza los valores de las variables

- <wait>: bloquea la ejecución durante un cierto período de tiempo

Las actividades estructuradas se asemejan a las estructuras de control de la programación convencional. Constituyen la parte orientada a bloques de **BPEL** e incluyen:

- <sequence>: conjunto de actividades que se ejecutarán en el orden indicado
- <comutador>: pares de condición-actividad a partir de los cuales la primera actividad con condición verdadera se ejecuta
- <pick>: lista de pares de evento-actividad. Cuando se produce el primer evento de la lista, se ejecuta la actividad correspondiente. El manejo de las condiciones de la ejecución no está especificado
- <while>: una actividad y condición. La actividad se ejecuta mientras la condición es verdad
- <flujo>: las actividades dentro de esta actividad se ejecutan en paralelo. Un flujo se completa cuando se completan todas las actividades paralelas

BPEL especifica además manejadores de eventos y fallos. Para cada manejador, se define un evento, un ámbito y la actividad que se realiza para manejar el evento.

El orden de ejecución dentro de un elemento <flow> se puede controlar usando elementos <link>. Esto define la naturaleza orientada a gráficos de **BPEL**. En consecuencia, los enlaces **BPEL** tienen un papel importante cuando los diagramas del proceso de negocio se transforman en procesos ejecutables. Existen ciertas restricciones en el uso de enlaces **BPEL**:

- los enlaces no pueden cruzar los límites de constructos de repetición, como <while>
- un <link> declarado en <flow> no puede crear un ciclo de control.

Conclusiones sobre BPEL En conclusión, **BPEL** es un lenguaje relativamente poderoso, la mejor opción posible en la actualidad para ejecutar procesos de negocio. En comparación con lenguajes de programación convencional, como Java, **BPEL** proporciona mecanismos potentes para representar interacciones típicas de los procesos de negocios, como transacciones a largo plazo, mensajería asincrónica y actividades paralelas, que en Java requeriría mucho más esfuerzo y líneas de código. La desventaja es su sintaxis restrictiva, lo que supone una fuente de problemas en las transformaciones **BPMN**.

Aunque **BPEL** es un estándar importante, carece de algunas características y complementos que requieren los lenguajes. Para la ejecución de procesos en redes comerciales electrónicas, se requiere un mayor soporte para los procesos de colaboración, para lo que es necesario usar “lenguajes de coreografía”, como WS-CDL (del inglés, Web Service Chorography Description Language) . El término coreografía se refiere a las colaboraciones entre partes que interactúan, es decir, el proceso global sobre el que cada participante debe estar de acuerdo. Por ejemplo, en base a las definiciones globales descritas en un documento WS-CDL, cada parte puede implementar soluciones que se ajusten a su rol en el proceso global.

Acrónimos

AD Activity Diagrama

AS Action Semantics

BD Base de Datos

BPDM Business Process Definition Metamodel

BPEL Business Process Execution Language

BPMN Business Process Model & Notation

CIM Computation Independent Model

CORBA Common Object Request Broker Architecture

D(S)ML Domain-Specific Modeling Language

DA Descripción Arquitectónica

EMF Eclipse Modeling Framework

GPML General Purpose Modeling Language

M2M Model to Model

M2T Model To Text

MBT Model Based Testing

MDA Model Driven Architecture

MDD Model Driven Development

MDE Model Driven Engineering

MDT Model Driven Testing

ML Modeling Language

MOF Meta Object Facility

MPS Meta Programming System

OCL Object Constraint Language

OMG Object Management Group

OO Object Oriented

PIM Platform Independent Model

PSM Platform Specific Model

QVT Query/View/Transformation

SDK Software Development Kit

TI Tecnología de la Información

WSDL Web Services Description Languages

XP Extreme Programming

xUML Executable UML

Bibliografía

- Havey, Michael. *Essential Business Process Modeling*. USA: O'Reilly Media, Inc., 2005.
- Koskela, Mika y Jyrki Haajanen. *Business Process Modeling and Execution. Research Notes 2407*. Informe técnico. 2007. <https://www.vttresearch.com/sites/default/files/pdf/tiedotteet/2007/T2407.pdf>.
- Mellor, Stephen J. y Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. USA: Addison-Wesley, 2002.
- Mellor, Stephen J., K. Scott, Uhl A. y Weise D. *MDA distilled: Principles of Model-Driven Architecture*. USA: Addison-Wesley Longman Publishing Co., Inc., 2004.
- Rodrigues da Silva, Alberto. «Model-Driven Engineering: A survey supported by a unified conceptual model». *Computer Languages, Systems & Structures* 20 (junio de 2015). <https://doi.org/10.1016/j.cl.2015.06.001>.
- Rozanski, Nick. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*. Addison-Wesley Professional, 2011. <https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/>.
- Warmer, Jos, Anneke Kleppe y Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. EE.UU.: Addison-Wesley Professional, 2003.

Test-1-Resuelto.pdf



Zukii



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

WUOLAH + BBVA

Llévate

15€

Al abrir tu Cuenta Online Sin Comisiones y hacer una compra superior a 15€.

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

¿Cómo? →



Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

17/5/22, 11:06

Cuestionario Tema 1: Revisión del intento

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah!
Tu que eres tan bonita

[Página Principal](#) / Mis cursos / [GRADUADO-A EN INGENIERÍA INFORMÁTICA \(2010\)-\(296\)](#)

/ [DESARROLLO DE SOFTWARE \(2122\)-296_11_3F_2122](#) / [Tema 1. Desarrollo utilizando patrones software](#) / [Cuestionario Tema 1](#)

Comenzado el martes, 17 de mayo de 2022, 10:55

Estado Finalizado

Finalizado en martes, 17 de mayo de 2022, 11:06

Tiempo 11 minutos 9 segundos
empleado

Calificación 9,67 de 10,00 (97%)

Pregunta 1

Parcialmente correcta

Se puntuó 1,67 sobre 2,00

Completa las tres máximas filosóficas de RoR o "meta-estilos arquitectónicos" más importantes:

[Fat] Model, [skinny] controllers, and [views]
[Convention] over [configuration]
Don't [repeat] yourself

dumb

Respuesta parcialmente correcta.

Ha seleccionado correctamente 5.

La respuesta correcta es:

Completa las tres máximas filosóficas de RoR o "meta-estilos arquitectónicos" más importantes:

[Fat] Model, [skinny] controllers, and [dumb] views

[Convention] over [configuration]

Don't [repeat] yourself

Pregunta 2

Correcta

Se puntuó 2,00 sobre 2,00

Arrastra la palabra adecuada sobre el texto para describir los cuatro elementos esenciales de un patrón de diseño:

- [Consecuencias de aplicarlo] ✓ : Los resultados y las ventajas e inconvenientes de aplicar el patrón.
- [Problema que trata de resolver] ✓ : describe cuándo y qué circunstancias se producen que requieran aplicar el patrón.
- [Solución propuesta] ✓ : los elementos que conforman el diseño, las relaciones entre los mismos, las responsabilidades y las colaboraciones, aunque sin hacerlo de forma concreta.
- [Nombre del patrón] ✓ : una o dos palabras usadas para describir un problema de diseño, sus soluciones y las consecuencias.

Respuesta correcta

La respuesta correcta es:

Arrastra la palabra adecuada sobre el texto para describir los cuatro elementos esenciales de un patrón de diseño:

[Consecuencias de aplicarlo]: Los resultados y las ventajas e inconvenientes de aplicar el patrón.

[Problema que trata de resolver]: describe cuándo y qué circunstancias se producen que requieran aplicar el patrón.

[Solución propuesta]: los elementos que conforman el diseño, las relaciones entre los mismos, las responsabilidades y las colaboraciones, aunque sin hacerlo de forma concreta.

[Nombre del patrón]: una o dos palabras usadas para describir un problema de diseño, sus soluciones y las consecuencias.



WUOLAH + BBVA

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Llévate

15€

1
Abre tu Cuenta Online sin comisiones ni condiciones

2
Haz una compra igual o superior a 15€ con tu nueva tarjeta

3
BBVA te devuelve 15€

¿Cómo? →

Cuéntame más



WUOLAH
+ BBVA

Pregunta 3

Correcta

Se puntuá 1,00 sobre 1,00

El patrón de diseño implementado en este código Ruby es el Delegator:

```
class User
  def born_on
    Date.new(1989, 9, 10)
  end
end

class UserDecorator < SimpleDelegator
  def birth_year
    born_on.year
  end
end

decorated_user = UserDecorator.new(User.new)
decorated_user.birth_year #=> 1989
decorated_user._getobj_ #=> #<User: ...>
```

Seleccione una:

- Verdadero
- Falso ✓

La respuesta correcta es 'Falso'

Pregunta 4

Correcta

Se puntuá 1,00 sobre 1,00

Indica un mecanismo de reusabilidad que favorecen los patrones de diseño como alternativa a la herencia (en especial, si la relación "ES-UN" no es tan obvia, pero a veces incluso cuando lo es)

Respuesta:

La composición



La respuesta correcta es: Composición

Que no te escriban poemas de amor cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

17/5/22, 11:06

Cuestionario Tema 1: Revisión del intento

Pregunta 5

Correcta

Se puntuó 4,00 sobre 4,00

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

Indica para cada patrón de diseño si es creacional, estructural o conductual.

Adaptador	Estructural	✓
Builder	Creacional	✓
Memento	Conductual	✓
Decorador	Estructural	✓
Visitante	Conductual	✓
Peso Ligero	Estructural	✓
Compuesto	Estructural	✓
Factoría abstracta	Creacional	✓
Orden (Command)	Conductual	✓
Intérprete	Conductual	✓
Proxy	Estructural	✓
Método Plantilla	Conductual	✓
Observer	Conductual	✓
Fachada	Estructural	✓
Cadena de Responsabilidad	Conductual	✓
Mediador	Conductual	✓
Puente	Estructural	✓
Iterador	Conductual	✓
Método Factoría	Creacional	✓
Estrategia	Conductual	✓
Singleton	Creacional	✓
State	Conductual	✓

Respuesta correcta

La respuesta correcta es:

Adaptador → Estructural,

Builder → Creacional,

Memento → Conductual,

Decorador → Estructural,

Visitante → Conductual,

Peso Ligero → Estructural,

Compuesto → Estructural,

Factoría abstracta → Creacional,

Orden (Command) → Conductual,

Intérprete → Conductual,

Proxy → Estructural,
Método Plantilla → Conductual,
Observer → Conductual,
Fachada → Estructural,
Cadena de Responsabilidad → Conductual,
Mediador → Conductual,
Puente → Estructural,
Iterador → Conductual,
Método Factoría → Creacional,
Estrategia → Conductual,
Singleton → Creacional,
State → Conductual

◀ GoF - Patrones

Ir a...

Patrón arquitectónico OO ►

Zukii

Examen-2-Resuelto.pdf



Zukii



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

WUOLAH + BBVA

Llévate

15€

Al abrir tu Cuenta Online Sin Comisiones y hacer una compra superior a 15€.

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

¿Cómo? →



THE BOOGEYMAN

YA EN CINES

[COMPRAR ENTRADAS](#)



6/6/22, 17:55

Examen segundo parcial: Revisión del intento

[Página Principal](#) / Mis cursos / [GRADUADO-A EN INGENIERÍA INFORMÁTICA \(2010\) \(296\)](#)
/ [DESARROLLO DE SOFTWARE \(2122\)-296_11_3F_2122](#) / [TEORÍA \(generalidades\)](#) / [Examen segundo parcial](#)

Comenzado el lunes, 6 de junio de 2022, 17:43

Estado Finalizado

Finalizado en lunes, 6 de junio de 2022, 17:55

Tiempo 12 minutos 18 segundos

empleado

Calificación Sin calificar aún

Pregunta 1

Correcta

Se puntuó 1,00 sobre 1,00

Arrastra la palabra adecuada sobre el texto para describir los cuatro elementos esenciales de un patrón de diseño:

Consecuencias de aplicarlo ✓ : Los resultados y las ventajas e inconvenientes de aplicar el patrón.

Problema que trata de resolver ✓ : describe cuándo y qué circunstancias se producen que requieran aplicar el patrón.

Solución propuesta ✓ : los elementos que conforman el diseño, las relaciones entre los mismos, las responsabilidades y las colaboraciones, aunque sin hacerlo de forma concreta.

Nombre del patrón ✓ : una o dos palabras usadas para describir un problema de diseño, sus soluciones y las consecuencias.

Respuesta correcta

La respuesta correcta es:

Arrastra la palabra adecuada sobre el texto para describir los cuatro elementos esenciales de un patrón de diseño:

[Consecuencias de aplicarlo]: Los resultados y las ventajas e inconvenientes de aplicar el patrón.

[Problema que trata de resolver]: describe cuándo y qué circunstancias se producen que requieran aplicar el patrón.

[Solución propuesta]: los elementos que conforman el diseño, las relaciones entre los mismos, las responsabilidades y las colaboraciones, aunque sin hacerlo de forma concreta.

[Nombre del patrón]: una o dos palabras usadas para describir un problema de diseño, sus soluciones y las consecuencias.

WUOLAH

Pregunta 2

Correcta

Se puntuá 1,00 sobre 1,00

Selecciona las características propias del mantenimiento de sistemas basados en componentes (CBS)

- a. Desplazamiento de los costes ✓
- b. Planificación más difícil ✓
- c. Habilidades específicas ✓
- d. Comunidad de usuarios pequeña
- e. Mantenimientos divididos ✓

Respuesta correcta

Las respuestas correctas son:

Mantenimientos divididos,

Habilidades específicas,

Desplazamiento de los costes,

Planificación más difícil

Pregunta 3

Correcta

Se puntuá 1,00 sobre 1,00

Escoge el concepto relacionado con las pruebas software que está siendo descrito

Los controladores ficticios (mocks) para pruebas de caja negra y los objetos ficticios (stubs) para hacer pruebas de unidad, son dos subtipos

Colección de software utilizado para la realización de pruebas software, especialmente las que se realizan de forma automática

Pruebas que tienen el cometido de descubrir errores asociados a la interfaz o interacción entre componentes

Colección de software y datos de prueba usados por desarrolladores para las pruebas de unidad

dobles de prueba

testware

pruebas de integración

arnés de prueba

Respuesta correcta

La respuesta correcta es:

Los controladores ficticios (mocks) para pruebas de caja negra y los objetos ficticios (stubs) para hacer pruebas de unidad, son dos subtipos → dobles de prueba,

Colección de software utilizado para la realización de pruebas software, especialmente las que se realizan de forma automática → testware,

Pruebas que tienen el cometido de descubrir errores asociados a la interfaz o interacción entre componentes → pruebas de integración,

Colección de software y datos de prueba usados por desarrolladores para las pruebas de unidad → arnés de prueba



WUOLAH + BBVA

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Llévate

15€

1
Abre tu Cuenta Online sin comisiones ni condiciones

2
Haz una compra igual o superior a 15€ con tu nueva tarjeta

3
BBVA te devuelve 15€

¿Cómo? →

Cuéntame más



WUOLAH
+ BBVA

Pregunta 4

Finalizado

Puntúa como 1,00

Explica en qué puede afectar de forma positiva algo de lo aprendido en la charla de Miguel Ángel López Montellano, de Fidesol, en tu actividad como desarrollador de software.

A desenmascarar muchos mitos pre establecidos en la industria como el de que la metodología ágil no siempre es la mejor solución o el que la "titulitis" que existe en España, no es tan real como dicen en este campo. También sobre la vida profesional de cara a decidir algunos aspectos que desconocíamos como la especialización VS capacidad de adaptación y rol VS tecnología

Pregunta 5

Finalizado

Puntúa como 1,00

Explica algo que cambiarías de la parte teórica de esta asignatura

Ya se hablaron algunas cosas en clase el penúltimo día de clase.

Pero algunas cosas mejorables creo que podría ser que los talleres deberían estar más testeados para evitar problemas, o el mejor seguimiento de los grupos/calificación individual en los grupos (aunque es mas sobre la asignatura en general), para evitar que no se aproveche del trabajo del otro y que alguien reciba menos calificación de la que debería



HAZTE UN BOWL A TU BOWL



6/6/22, 17:55

Examen segundo parcial: Revisión del intento

Pregunta 6

Finalizado

Puntúa como 1,00

Explica en qué puede afectar de forma positiva la experiencia en la parte teórica de esta asignatura en tu actividad como desarrollador de software

Las prácticas en aprender nuevas tecnologías y saber cómo aplicarlas.

Respecto a la pregunta, por ejemplo, a ver algunos ejemplos de aplicación de patrones de diseño o a saber planificar el desarrollo del software y sus pruebas...

Pregunta 7

Correcta

Se puntuó 1,00 sobre 1,00

Asocia cada definición con el término que la define

Modelo

Abstracción de un sistema bajo estudio



Sistema

Conjunto ordenado de normas y procedimientos que regulan el funcionamiento de un grupo o colectividad



Meta-metamodelo

Modelo que describe el lenguaje de modelado de un metamodelo



Metamodelo

Modelo que define la estructura de un lenguaje de modelado



Respuesta correcta

La respuesta correcta es:

Modelo → Abstracción de un sistema bajo estudio,

Sistema → Conjunto ordenado de normas y procedimientos que regulan el funcionamiento de un grupo o colectividad,

Meta-metamodelo → Modelo que describe el lenguaje de modelado de un metamodelo,

Metamodelo → Modelo que define la estructura de un lenguaje de modelado

Pregunta 8

Correcta

Se puntuó 1,00 sobre 1,00

Elije entre los requisitos siguientes aquellos considerados perspectivas muy importantes en proyectos de gran envergadura, según Rozansky.

- a. Fácil de usar ✓
- b. Internacionalizable ✓
- c. Sostenible
- d. Accesible ✓
- e. Seguro ✓

Respuesta correcta

Las respuestas correctas son:

Accesible,

Fácil de usar,

Internacionalizable,

Seguro

Pregunta 9

Correcta

Se puntuá 1,00 sobre 1,00

Primera ley: de ✓ .-A menos que un sistema se modifique continuamente para satisfacer las necesidades emergentes de los usuarios, el sistema se vuelve cada vez menos útil.

Segunda ley: de ✓ .- A menos que se realice un trabajo adicional para reducir explícitamente la complejidad de un sistema, el sistema se volverá cada vez más complejo y menos estructurado debido a los cambios relacionados con el mantenimiento.

Tercera ley: de ✓ .-El proceso de evolución puede parecer aleatorio en vistas puntuales de tiempo y de espacio pero a lo largo del tiempo, se aprecia a nivel estadístico una patrón de conservación cíclico, con tendencias bien definidas a largo plazo. Así, las medidas del nivel de mantenimiento requerido por los distintos productos y procesos, que se producen durante la evolución, siguen una distribución cercana a la normal.

Respuesta correcta

La respuesta correcta es:

Primera ley: de [Cambio continuo]:.-A menos que un sistema se modifique continuamente para satisfacer las necesidades emergentes de los usuarios, el sistema se vuelve cada vez menos útil.

Segunda ley: de [Entropía].- A menos que se realice un trabajo adicional para reducir explícitamente la complejidad de un sistema, el sistema se volverá cada vez más complejo y menos estructurado debido a los cambios relacionados con el mantenimiento.

Tercera ley: de [Autorregulación].-El proceso de evolución puede parecer aleatorio en vistas puntuales de tiempo y de espacio pero a lo largo del tiempo, se aprecia a nivel estadístico una patrón de conservación cíclico, con tendencias bien definidas a largo plazo. Así, las medidas del nivel de mantenimiento requerido por los distintos productos y procesos, que se producen durante la evolución, siguen una distribución cercana a la normal.



HAZTE UN BOWL A TU BOWL



6/6/22, 17:55

Examen segundo parcial: Revisión del intento

Pregunta 10

Correcta

Se puntuó 1,00 sobre 1,00

Asocia las distintas clasificaciones de los lenguajes de modelado con el criterio de clasificación utilizado

Según dominio de aplicación

De uso general y de uso específico



Según nivel de abstracción

Computation Independent (CIM), platform independent (PIM) y platform specific (PSM)



Según punto de vista funcional

Estático y dinámico



Respuesta correcta

La respuesta correcta es:

Según dominio de aplicación → De uso general y de uso específico,

Según nivel de abstracción → Computation Independent (CIM), platform independent (PIM) y platform specific (PSM),

Según punto de vista funcional → Estático y dinámico

◀ Examen primer parcial (no presencial)

Ir a...

Asistencia Prácticas ►



ZUKII

Test-3-Resuelto.pdf



Zukii



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

WUOLAH + BBVA

Llévate

15€

Al abrir tu Cuenta Online Sin Comisiones y hacer una compra superior a 15€.

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

¿Cómo? →



Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

24/5/22, 17:05

Cuestionario Tema 3: Revisión del intento

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

[Página Principal](#) / Mis cursos / [GRADUADO-A EN INGENIERÍA INFORMÁTICA \(2010\)-\(296\)](#)

/ [DESARROLLO DE SOFTWARE \(2122\)-296_11_3F_2122](#) / [Tema 3. Validación de Software](#) / [Cuestionario Tema 3](#)

Comenzado el martes, 24 de mayo de 2022, 16:48

Estado Finalizado

Finalizado en martes, 24 de mayo de 2022, 17:05

Tiempo empleado 16 minutos 48 segundos

Calificación 8,00 de 10,00 (80%)

Pregunta 1

Correcta

Se puntuó 2,00 sobre 2,00

Elige las pruebas que son consideradas dentro del factor operativo, según McCall

- a. Pruebas de correctitud ✓
- b. Pruebas de fiabilidad ✓
- c. Pruebas de estrés ✓
- d. Pruebas de portabilidad
- e. Pruebas de facilidad de testeo
- f. Pruebas de seguridad ✓

Respuesta correcta

Las respuestas correctas son:

Pruebas de correctitud,

Pruebas de fiabilidad,

Pruebas de estrés,

Pruebas de seguridad

Pregunta 2

Correcta

Se puntuá 2,00 sobre 2,00

Escoge el concepto relacionado con las pruebas software que está siendo descrito

Colección de software utilizado para la realización de pruebas software, especialmente las que se realizan de forma automática

Los controladores ficticios (mocks) para pruebas de caja negra y los objetos ficticios (stubs) para hacer pruebas de unidad, son dos subtipos

Colección de software y datos de prueba usados por desarrolladores para las pruebas de unidad

Pruebas que tienen el cometido de descubrir errores asociados a la interfaz o interacción entre componentes

testware

dobles de prueba

arnés de prueba

pruebas de integración

Respuesta correcta

La respuesta correcta es:

Colección de software utilizado para la realización de pruebas software, especialmente las que se realizan de forma automática → testware,

Los controladores ficticios (mocks) para pruebas de caja negra y los objetos ficticios (stubs) para hacer pruebas de unidad, son dos subtipos → dobles de prueba,

Colección de software y datos de prueba usados por desarrolladores para las pruebas de unidad → arnés de prueba,

Pruebas que tienen el cometido de descubrir errores asociados a la interfaz o interacción entre componentes → pruebas de integración

Pregunta 3

Correcta

Se puntuá 2,00 sobre 2,00

La **verificación** ✓ es el proceso de evaluar si el software cumple con los requisitos especificados al inicio.

La **validación** ✓ es el proceso de evaluar si el software cumple de forma completa y correcta con su propósito original, es decir, hasta qué punto el producto satisface la intencionalidad con la que fue concebido por el cliente, aun cuando no se hubiera reflejado bien en las especificaciones.

La **cualificación** ✓ es el proceso de evaluar si el software cumple de forma completa y correcta con todos sus aspectos operativos que faciliten el mantenimiento.

Respuesta correcta

La respuesta correcta es:

La [verificación] es el proceso de evaluar si el software cumple con los requisitos especificados al inicio.

La [validación] es el proceso de evaluar si el software cumple de forma completa y correcta con su propósito original, es decir, hasta qué punto el producto satisface la intencionalidad con la que fue concebido por el cliente, aun cuando no se hubiera reflejado bien en las especificaciones.

La [cualificación] es el proceso de evaluar si el software cumple de forma completa y correcta con todos sus aspectos operativos que faciliten el mantenimiento.



WUOLAH + BBVA

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Llévate

15€

1
Abre tu Cuenta Online sin comisiones ni condiciones

2
Haz una compra igual o superior a 15€ con tu nueva tarjeta

3
BBVA te devuelve 15€

¿Cómo? →

Cuéntame más



WUOLAH
+ BBVA

Pregunta 4

Incorrecta

Se puntuó 0,00 sobre 2,00

Ordena las pruebas según el momento en el que se realiza, empezando por las que se realizan primero:

- [1] Pruebas de sistema
- [4] Pruebas de unidad
- [3] Pruebas beta
- [6] Pruebas de integración
- [2] Pruebas de aceptación
- [5] Pruebas de componentes

Respuesta incorrecta.

La respuesta correcta es:

Ordena las pruebas según el momento en el que se realiza, empezando por las que se realizan primero:

- [4] Pruebas de sistema
- [1] Pruebas de unidad
- [6] Pruebas beta
- [3] Pruebas de integración
- [5] Pruebas de aceptación
- [2] Pruebas de componentes

Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

24/5/22, 17:05

Cuestionario Tema 3: Revisión del intento

Pregunta 5

Correcta

Se puntuó 2,00 sobre 2,00

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

Elige de cada descripción si se refiere a garantía o a control de calidad

Es competencia de todo el personal

Garantía de calidad ✓

Se concentra en el producto en sí

Control de calidad ✓

Se concentra en el proceso de producción
de un producto

Garantía de calidad ✓

Es una tarea atribuida a un equipo dedicado a ella

Control de calidad ✓

Respuesta correcta

La respuesta correcta es:

Es competencia de todo el personal → Garantía de calidad,

Se concentra en el producto en sí → Control de calidad,

Se concentra en el proceso de producción
de un producto → Garantía de calidad,

Es una tarea atribuida a un equipo dedicado a ella → Control de calidad

◀ Tema 3 presentación

Ir a...

Tema 4 ▶

Zukii

Test-4-Resuelto.pdf



Zukii



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

WUOLAH + BBVA

Llévate

15€

Al abrir tu Cuenta Online Sin Comisiones y hacer una compra superior a 15€.

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

¿Cómo? →



Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

24/5/22, 17:17

Cuestionario Tema 4: Revisión del intento

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

[Página Principal](#) / Mis cursos / [GRADUADO-A EN INGENIERÍA INFORMÁTICA \(2010\)-\(296\)](#)

/ [DESARROLLO DE SOFTWARE \(2122\)-296_11_3F_2122](#) / [Tema 4. Mantenimiento y Evolución del Software](#) / [Cuestionario Tema 4](#)

Comenzado el martes, 24 de mayo de 2022, 17:06

Estado Finalizado

Finalizado en martes, 24 de mayo de 2022, 17:17

Tiempo 10 minutos 52 segundos
empleado

Calificación 10,00 de 10,00 (100%)

Pregunta 1

Correcta

Se puntuó 2,00 sobre 2,00

Todo proceso de reingeniería tiene tres etapas: La primera etapa es la ingeniería inversa ✓ ,

La etapa central es la especificación del nuevo producto ✓ y La etapa final es la ingeniería directa ✓ .

La primera etapa es la ingeniería directa

La etapa final es la ingeniería inversa

La etapa central es la ingeniería inversa

La etapa final es la especificación del nuevo producto

La etapa central es la ingeniería directa

Respuesta correcta

La respuesta correcta es:

Todo proceso de reingeniería tiene tres etapas: [La primera etapa es la ingeniería inversa], [La etapa central es la especificación del nuevo producto] y [La etapa final es la ingeniería directa].



WUOLAH

Pregunta 2

Correcta

Se puntuá 2,00 sobre 2,00

Selecciona las características propias del mantenimiento de sistemas basados en componentes (CBS)

- a. Habilidades específicas ✓
- b. Planificación más difícil ✓
- c. Comunidad de usuarios pequeña
- d. Mantenimientos divididos ✓
- e. Desplazamiento de los costes ✓

Respuesta correcta

Las respuestas correctas son:

Mantenimientos divididos,

Habilidades específicas,

Desplazamiento de los costes,

Planificación más difícil

Pregunta 3

Correcta

Se puntuá 2,00 sobre 2,00

Hemos decidido migrar nuestra aplicación del framework RoR basado en Ruby al framework Keymaker basado en Java y Neo4j, con un DBMS basado en grafos . ¿Qué tres formas de hacerlo tenemos?

Migración directa

Migramos primero la base de datos relacional. Mientras vamos actualizando toda la aplicación, hacemos uso de los dos sistemas.



Base de datos compuesta

Migramos poco a poco pero simultáneamente la base de datos y el software necesitando una doble pasarela para mantener la integridad de los datos.



Migración inversa

Convertimos todo el código al nuevo sistema usando la base de datos original. Mientras que lo hacemos usamos una pasarela para acceder a la nueva base de datos.



Respuesta correcta

La respuesta correcta es:

Migración directa → Migramos primero la base de datos relacional. Mientras vamos actualizando toda la aplicación, hacemos uso de los dos sistemas, accediendo los antiguos a la nueva base de datos mediante una pasarela.

Base de datos compuesta → Migramos poco a poco pero simultáneamente la base de datos y el software necesitando una doble pasarela para mantener la integridad de los datos.

Migración inversa → Convertimos todo el código al nuevo sistema usando la base de datos original. Mientras que lo hacemos usamos una pasarela para que el nuevo código acceda temporalmente a la base de datos heredada. Al final, convertimos la base de datos.





WUOLAH + BBVA

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Llévate

15€

1
Abre tu Cuenta Online sin comisiones ni condiciones

2
Haz una compra igual o superior a 15€ con tu nueva tarjeta

3
BBVA te devuelve 15€

¿Cómo? →

Cuéntame más



WUOLAH
+ BBVA

Pregunta 4

Correcta

Se puntuó 2,00 sobre 2,00

Señala las características del código que son muestra de hediondez

- a. Métodos muy largos ✓
- b. Clases con muy pocos métodos
- c. Encadenamiento de mensajes ✓
- d. Métodos con muchos parámetros ✓
- e. Código repetido ✓

Respuesta correcta

Las respuestas correctas son:

Métodos con muchos parámetros,

Métodos muy largos,

Encadenamiento de mensajes,

Código repetido

Que no te escriban poemas de amor cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

24/5/22, 17:17

Cuestionario Tema 4: Revisión del intento

Pregunta 5

Correcta

Se puntuó 2,00 sobre 2,00

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

Primera ley: de Cambio continuo ✓ :-A menos que un sistema se modifique continuamente para satisfacer las necesidades emergentes de los usuarios, el sistema se vuelve cada vez menos útil.

Segunda ley: de Entropía ✓ .- A menos que se realice un trabajo adicional para reducir explícitamente la complejidad de un sistema, el sistema se volverá cada vez más complejo y menos estructurado debido a los cambios relacionados con el mantenimiento.

Tercera ley: de Autorregulación ✓ .-El proceso de evolución puede parecer aleatorio en vistas puntuales de tiempo y de espacio pero a lo largo del tiempo, se aprecia a nivel estadístico una autorregulación (ley de conservación) cíclica, con tendencias bien definidas a largo plazo. Así, las medidas del nivel de mantenimiento requerido por los distintos productos y procesos, que se producen durante la evolución, siguen una distribución cercana a la normal.

Respuesta correcta

La respuesta correcta es:

Primera ley: de [Cambio continuo]:-A menos que un sistema se modifique continuamente para satisfacer las necesidades emergentes de los usuarios, el sistema se vuelve cada vez menos útil.

Segunda ley: de [Entropía].- A menos que se realice un trabajo adicional para reducir explícitamente la complejidad de un sistema, el sistema se volverá cada vez más complejo y menos estructurado debido a los cambios relacionados con el mantenimiento.

Tercera ley: de [Autorregulación].-El proceso de evolución puede parecer aleatorio en vistas puntuales de tiempo y de espacio pero a lo largo del tiempo, se aprecia a nivel estadístico una autorregulación (ley de conservación) cíclica, con tendencias bien definidas a largo plazo. Así, las medidas del nivel de mantenimiento requerido por los distintos productos y procesos, que se producen durante la evolución, siguen una distribución cercana a la normal.

◀ Tema 4 - Presentación

Ir a ...



Tema 5 ▶

WUOLAH

Zukii

Test-5-Resuelto.pdf



Zukii



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

WUOLAH + BBVA

Llévate

15€

Al abrir tu Cuenta Online Sin Comisiones y hacer una compra superior a 15€.

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

¿Cómo? →



Que no te escriban poemas de amor cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

24/5/22, 17:26

Cuestionario tema 5: Revisión del intento

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah!
Tu que eres tan bonita

[Página Principal](#) / Mis cursos / [GRADUADO-A EN INGENIERÍA INFORMÁTICA \(2010\)-\(296\)](#)

/ [DESARROLLO DE SOFTWARE \(2122\)-296_11_3F_2122](#) / [Tema 5. Desarrollo dirigido por modelos](#) / [Cuestionario tema 5](#)

Comenzado el martes, 24 de mayo de 2022, 17:18

Estado Finalizado

Finalizado en martes, 24 de mayo de 2022, 17:26

Tiempo empleado 8 minutos 51 segundos

Calificación 9,75 de 10,00 (98%)

Pregunta 1

Correcta

Se puntuó 1,00 sobre 1,00

Asocia las distintas clasificaciones de los lenguajes de modelado con el criterio de clasificación utilizado

Según nivel de abstracción

Computation Independent (CIM), platform independent (PIM) y platform specific (PSM)



Según dominio de aplicación

De uso general y de uso específico



Según punto de vista funcional

Estático y dinámico



Respuesta correcta

La respuesta correcta es:

Según nivel de abstracción → Computation Independent (CIM), platform independent (PIM) y platform specific (PSM),

Según dominio de aplicación → De uso general y de uso específico,

Según punto de vista funcional → Estático y dinámico

Pregunta 2

Correcta

Se puntuó 1,00 sobre 1,00

Asocia los distintos enfoques MDE con el nivel de abstracción al que pertenecen

Nivel bajo (enfoques concretos)

Factoría software Web Service y Sistemas Interactivos eXtensibles para móviles (XIS-Mobile)



Nivel alto

MDA, MDD, MDT



Nivel medio

Factorías software de Microsoft y entorno de trabajo de modelado de Eclipse



Respuesta correcta

La respuesta correcta es:

Nivel bajo (enfoques concretos) → Factoría software Web Service y Sistemas Interactivos eXtensibles para móviles (XIS-Mobile),

Nivel alto → MDA, MDD, MDT,

Nivel medio → Factorías software de Microsoft y entorno de trabajo de modelado de Eclipse

Pregunta 3

Parcialmente correcta

Se puntuá 0,75 sobre 1,00

Selecciona entre los siguientes apartados, aquéllos que describan ejemplos de MDE (Model Driven Architecture)

- a. Microsoft Software Factories ✓
- b. Model Driven Architecture (MDA) del OMG ✓
- c. Eclipse Modeling Framework
- d. Model Driven Testing ✓
- e. Contenedores software para despliegue de aplicaciones (DOCKER)

Respuesta parcialmente correcta.

Ha seleccionado correctamente 3.

Las respuestas correctas son:

Model Driven Architecture (MDA) del OMG,

Microsoft Software Factories,

Model Driven Testing,

Eclipse Modeling Framework

Pregunta 4

Correcta

Se puntuá 1,00 sobre 1,00

UML es un ejemplo de meta-metamodelo, propio de la última capa (M3) del estándar Meta Object Facility (MOF) de MDA

Seleccione una:

- Verdadero
- Falso ✓

La respuesta correcta es 'Falso'

Pregunta 5

Correcta

Se puntuá 1,00 sobre 1,00

El compilador de Lisp está escrito en C

Seleccione una:

- Verdadero
- Falso ✓

La respuesta correcta es 'Falso'



WUOLAH + BBVA

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Llévate

15€

1
Abre tu Cuenta Online sin comisiones ni condiciones

2
Haz una compra igual o superior a 15€ con tu nueva tarjeta

3
BBVA te devuelve 15€

¿Cómo? →

Cuéntame más



WUOLAH
+ BBVA

Pregunta 6

Correcta

Se puntuá 1,00 sobre 1,00

La metaclasses de una clase Ruby es un ejemplo de metamodelo

Seleccione una:

- Verdadero ✓
- Falso

La respuesta correcta es 'Verdadero'

Pregunta 7

Correcta

Se puntuá 1,00 sobre 1,00

Los marcadores que ayudan a mantener el software desarrollado con metodologías ágiles no evitan la necesidad de producir documentación de alto nivel

Seleccione una:

- Verdadero ✓
- Falso

La respuesta correcta es 'Verdadero'

Pregunta 8

Correcta

Se puntuá 1,00 sobre 1,00

Asocia cada definición con el término que la define

Sistema	Conjunto ordenado de normas y procedimientos que regulan el funcionamiento de un grupo o colectividad
Meta-metamodelo	Modelo que describe el lenguaje de modelado de un metamodelo
Modelo	Abstracción de un sistema bajo estudio
Metamodelo	Modelo que define la estructura de un lenguaje de modelado

Respuesta correcta

La respuesta correcta es:

Sistema → Conjunto ordenado de normas y procedimientos que regulan el funcionamiento de un grupo o colectividad,

Meta-metamodelo → Modelo que describe el lenguaje de modelado de un metamodelo,

Modelo → Abstracción de un sistema bajo estudio,

Metamodelo → Modelo que define la estructura de un lenguaje de modelado

Que no te escriban poemas de amor cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

24/5/22, 17:26

Cuestionario tema 5: Revisión del intento

Pregunta 9

Correcta

Se puntuó 1,00 sobre 1,00

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

RoR tiene una herramienta de transformación (puente) PSM-PSM implícita que permite definir entidades como objetos Ruby y como tablas en un SGBD de forma simultánea y automática

Seleccione una:

- Verdadero ✓
 Falso

La respuesta correcta es 'Verdadero'

Pregunta 10

Correcta

Se puntuó 1,00 sobre 1,00

El enfoque MD permite ser completamente productivos desde las primeras fases del ciclo de vida

Seleccione una:

- Verdadero ✓
 Falso

La respuesta correcta es 'Verdadero'

◀ Tema 5 - Presentación

Ir a ...

P1 ►



ZUKII

Test-2-Resuelto.pdf



Zukii



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

WUOLAH + BBVA

Llévate

15€

Al abrir tu Cuenta Online Sin Comisiones y hacer una compra superior a 15€.

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

¿Cómo? →



Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

17/5/22, 11:19

Cuestionario Tema 2: Revisión del intento

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

[Página Principal](#) / Mis cursos / [GRADUADO-A EN INGENIERÍA INFORMÁTICA \(2010\) \(296\)](#)

/ [DESARROLLO DE SOFTWARE \(2122\)-296_11_3F_2122](#) / [Tema 2. Arquitecturas de Software](#) / [Cuestionario Tema 2](#)

Comenzado el martes, 17 de mayo de 2022, 11:06

Estado Finalizado

Finalizado en martes, 17 de mayo de 2022, 11:19

Tiempo empleado 12 minutos 8 segundos

Calificación 4,33 de 10,00 (43%)

Pregunta 1

Parcialmente correcta

Se puntuó 0,67 sobre 2,00

Características de SOA y subtipos

- | | |
|------|---|
| SOA | Subtipo de arquitectura orientada al servicio que permite cualquier formato de transmisión de datos siempre que haya uniformidad. |
| SOAP | Subtipo de arquitectura orientada al servicio que usa XML para transferencia de datos. |
| REST | Arquitectura orientada al servicio: subtipo de la arquitectura Cliente-Servidor que desacopla clientes y servidores. |
- ✗
- ✓
- ✗

Respuesta parcialmente correcta.

Ha seleccionado correctamente 1.

La respuesta correcta es:

SOA → Arquitectura orientada al servicio: subtipo de la arquitectura Cliente-Servidor que desacopla clientes y servidores,

SOAP → Subtipo de arquitectura orientada al servicio que usa XML para transferencia de datos,

REST → Subtipo de arquitectura orientada al servicio que permite cualquier formato de transmisión de datos siempre que haya uniformidad

Pregunta 2

Parcialmente correcta

Se puntuó 0,67 sobre 2,00

Diferencias HTTP-CRUD-REST

HTTP	Arquitectura C/S sin estado/cacheable/por capas/interfaz unifome	
CRUD	Funciones básicas para mantenimiento de los datos persistente	
REST	Protocolo de transferencia no obligatorio, aunque el más común, en sistemas RESTful	

Respuesta parcialmente correcta.

Ha seleccionado correctamente 1.

La respuesta correcta es:

HTTP → Protocolo de transferencia no obligatorio, aunque el más común, en sistemas RESTful,

CRUD → Funciones básicas para mantenimiento de los datos persistente,

REST → Arquitectura C/S sin estado/cacheable/por capas/interfaz unifome

Pregunta 3

Correcta

Se puntuó 1,00 sobre 1,00

Las perspectivas arquitectónicas describen la arquitectura del software que será desarrollado de forma que se garantice el cumplimiento de los requisitos funcionales.

Seleccione una:

- Verdadero
 Falso

La respuesta correcta es 'Falso'

Pregunta 4

Incorrecta

Se puntuó 0,00 sobre 1,00

Una aplicación RESTful es aquella que cumple perfectamente con el estilo arquitectónico REST

Seleccione una:

- Verdadero
 Falso

La respuesta correcta es 'Verdadero'



WUOLAH + BBVA

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Llévate

15€

1
Abre tu Cuenta Online sin comisiones ni condiciones

2
Haz una compra igual o superior a 15€ con tu nueva tarjeta

3
BBVA te devuelve 15€

¿Cómo? →

Cuéntame más



WUOLAH
+ BBVA

Pregunta 5

Incorrecta

Se puntuó 0,00 sobre 2,00

Elige una función de HTTP usada generalmente para implantar REST que sea idempotente y describe lo que hace

Respuesta:

GET, hacer una petición para recibir datos desde el servidor



La respuesta correcta es: PUT

Pregunta 6

Correcta

Se puntuó 2,00 sobre 2,00

Elige entre los requisitos siguientes aquellos considerados perspectivas muy importantes en proyectos de gran envergadura, según Rozansky.

- a. Fácil de usar ✓
- b. Accesible ✓
- c. Seguro ✓
- d. Internacionalizable ✓
- e. Sostenible ✓

Respuesta correcta

Las respuestas correctas son:

Accesible,

Fácil de usar,

Internacionalizable,

Seguro



◀ Tema 2 - Presentación

Ir a...

S5 ►

**Que no te escriban poemas de amor
cuando terminen la carrera ►►►►►**



WUOLAH

(a nosotros por suerte nos pasa)

@Zukii on Wuolah

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

ZUKII

Solucion-Cuestionarios-DS.pdf



jnabre_58



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

WUOLAH + BBVA

Llévate

15€

Al abrir tu Cuenta Online Sin Comisiones y hacer una compra superior a 15€.

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

¿Cómo? →



Cuánto más



Invita a otros estudiantes, crea contenido y gana los premios que te alegrarán el verano

Jnabre58

Cuestionarios Desarrollo de Software. Curso 22/23.

Cuestionario 1

Pregunta 1
Respuesta guardada
Puntúa como 1,00
 Marcar pregunta

Completa las tres máximas filosóficas de RoR o "meta-estilos arquitectónicos" más importantes:
Fat Model, skinny controllers, and dumb views
Convention over configuration
Don't repeat yourself

Elige una función de HTTP usada generalmente para implantar REST que sea idempotente y describe lo que hace

Respuesta: GET: obtiene una representación de un recurso ✖

La respuesta correcta es: PUT

Pregunta 3
Sin responder aún
Puntúa como 2,00
 Marcar pregunta

Diferencias HTTP-CRUD-REST

HTTP	Protocolo de transferencia no obligatorio, aunque el más común, en sistemas RESTful
REST	Arquitectura C/S sin estado/cacheable/por capas/interfaz unifome
CRUD	Funciones básicas para mantenimiento de los datos persistentes

Pregunta 4
Sin responder aún
Puntúa como 1,00
 Marcar pregunta

El patrón de diseño implementado en este código Ruby es el Delegator:

```
class User
  def born_on
    Date.new(1989, 9, 10)
  end
end

class UserDecorator < SimpleDelegator
  def birth_year
    born_on.year
  end
end

decorated_user = UserDecorator.new(User.new)
decorated_user.birth_year #=> 1989
decorated_user._getobj_ #=> #<User: ...>
```

Seleccione una:

Verdadero
 Falso

Desarrollo de Software 22/23

WUOLAH

Hstos el
15/06/2023

Pregunta 5
Sin responder aún
Puntúa como 1,00
 Marcar pregunta

Indica para cada patrón de diseño si es creacional, estructural o conductual.

Visitante	Conductual
Cadena de Responsabilidad	Conductual
Intérprete	Conductual
Puente	Estructural
Compuesto	Estructural
Método Factoría	Creacional
Factoría abstracta	Creacional
Proxy	Estructural
Iterador	Conductual
Estrategia	Conductual
Orden (Command)	Conductual
Adaptador	Estructural
Observer	Conductual
Singleton	Creacional
Mediador	Conductual
Peso Ligero	Estructural
State	Conductual
Fachada	Estructural
Decorador	Estructural
Builder	Creacional
Método Plantilla	Conductual
Memento	Conductual



WUOLAH + BBVA

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Llévate

15€

1
Abre tu Cuenta Online sin comisiones ni condiciones

2
Haz una compra igual o superior a 15€ con tu nueva tarjeta

3
BBVA te devuelve 15€

¿Cómo? →

Cuéntame más



WUOLAH
+ BBVA

Pregunta 6
Parcialmente correcta
Se puntuó 0,33 sobre 1,00
P Marcar pregunta

Características de SOA y subtipos

REST	Subtipo de arquitectura orientada al servicio que permite cualquier formato de transmisión de datos siempre que haya uniformidad.
SOAP	Arquitectura orientada al servicio: subtipo de la arquitectura Cliente-Servidor que desacopla clientes y servidores.
SOA	Subtipo de arquitectura orientada al servicio que usa XML para transferencia de datos.

✓ ✗ ✗

Respuesta parcialmente correcta.
Ha seleccionado correctamente 1.
La respuesta correcta es:
REST → Subtipo de arquitectura orientada al servicio que permite cualquier formato de transmisión de datos siempre que haya uniformidad.
SOAP → Subtipo de arquitectura orientada al servicio que usa XML para transferencia de datos.
SOA → Arquitectura orientada al servicio: subtipo de la arquitectura Cliente-Servidor que desacopla clientes y servidores

Pregunta 7
Sin responder aún
Puntúa como 1,00
P Marcar pregunta

Indica un mecanismo de reusabilidad que favorecen los patrones de diseño como alternativa a la herencia (en especial, si la relación "ES-UN" no es tan obvia, pero a veces incluso cuando lo es)

Respuesta: La composición

Pregunta 9
Sin responder aún
Puntúa como 1,00
P Marcar pregunta

Arrastra la palabra adecuada sobre el texto para describir los cuatro elementos esenciales de un patrón de diseño:

Consecuencias de aplicarlo	: Los resultados y las ventajas e inconvenientes de aplicar el patrón.
Problema que trata de resolver	: describe cuándo y qué circunstancias se producen que requieran aplicar el patrón.
Solución propuesta	: los elementos que conforman el diseño, las relaciones entre los mismos, las responsabilidades y las colaboraciones, aunque sin hacerlo de forma concreta.
Nombre del patrón	: una o dos palabras usadas para describir un problema de diseño, sus soluciones y las consecuencias.

Cuestionario 2

<p>Pregunta 1 Sin responder aún Puntúa como 2,00 <input type="button" value="Marcar pregunta"/></p>	<p>Todo proceso de reingeniería tiene tres etapas: La primera etapa es la ingeniería inversa, La etapa central es la especificación del nuevo producto y La etapa final es la ingeniería directa.</p>
<p>Pregunta 2 Sin responder aún Puntúa como 2,00 <input type="button" value="Marcar pregunta"/></p>	<p>Selecciona las características propias del mantenimiento de sistemas basados en componentes (CBS)</p> <p><input checked="" type="checkbox"/> a. Planificación más difícil <input checked="" type="checkbox"/> b. Desplazamiento de los costes <input checked="" type="checkbox"/> c. Habilidades específicas <input checked="" type="checkbox"/> d. Mantenimientos divididos <input type="checkbox"/> e. Comunidad de usuarios pequeña</p>
<p>Pregunta 3 Sin responder aún Puntúa como 2,00 <input type="button" value="Marcar pregunta"/></p>	<p>Señala las características del código que son muestra de hediondez</p> <p><input type="checkbox"/> a. Clases con muy pocos métodos <input checked="" type="checkbox"/> b. Encadenamiento de mensajes <input checked="" type="checkbox"/> c. Métodos con muchos parámetros <input checked="" type="checkbox"/> d. Código repetido <input checked="" type="checkbox"/> e. Métodos muy largos</p>



Invita a otros estudiantes, crea contenido y gana los premios que te alegrarán el verano

Jnabre58



participa
aqui

Pregunta 4
Sin responder aún
Puntúa como 2,00
 Marcar pregunta

Primera ley: de **Cambio continuo** :-A menos que un sistema se modifique continuamente para satisfacer las necesidades emergentes de los usuarios, el sistema se vuelve cada vez menos útil.

Segunda ley: de **Entropía** .- A menos que se realice un trabajo adicional para reducir explícitamente la complejidad de un sistema, el sistema se volverá cada vez más complejo y menos estructurado debido a los cambios relacionados con el mantenimiento.

Tercera ley: de **Autorregulación** .-El proceso de evolución puede parecer aleatorio en vistas puntuales de tiempo y de espacio pero a lo largo del tiempo, se aprecia a nivel estadístico una patrón de conservación cíclico, con tendencias bien definidas a largo plazo. Así, las medidas del nivel de mantenimiento requerido por los distintos productos y procesos, que se producen durante la evolución, siguen una distribución cercana a la normal.

Pregunta 5
Sin responder aún
Puntúa como 2,00
 Marcar pregunta

Hemos decidido migrar nuestra aplicación del framework RoR basado en Ruby al framework Keymaker basado en Java y Neo4j, con un DBMS basado en grafos . ¿Qué tres formas más importantes de hacerlo tenemos?

Base de datos compuesta **Migramos poco a poco pero simultáneamente la base de datos y el software necesitando una doble pasarela para acceder a ambos sistemas.**

Migración directa **Migramos primero la base de datos relacional. Mientras vamos actualizando toda la aplicación, hacemos uso de la migración directa.**

Migración inversa **Convertimos todo el código al nuevo sistema usando la base de datos original. Mientras que lo hacemos usamos la migración inversa.**

Premio
WUOLAH 2023

Hstas el
15/06/2023

Desarrollo de Software 22/23

WUOLAH

Cuestionario 3

Pregunta 1
Sin responder aún
Puntúa como 1,00
Marcar pregunta

Ingredientes básicos del estándar IEEE P171 de descripción arquitectónica del software. Completa el texto:

Conjunto de normas de **definición de términos** tales como descripción arquitectónica, vista arquitectónica o punto de vista arquitectónico

Marco conceptual que sitúa estos términos en el contexto de los múltiples usos posibles de DAs para la construcción, análisis y evolución de sistemas

Conjunto de **requerimientos** sobre una DA de un sistema

El estándar IEEE P171 es **independiente** de la notación

Pregunta 2
Parcialmente correcta
Se puntuó 0,67 sobre 1,00
Marcar pregunta

Relaciona las distintas propiedades no funcionales de la arquitectura software con su definición

Facilidad con la que se pueden mover elementos entre distintos módulos o subsistemas, lo cual implica también cambios en los clientes de los mismos

Reestructuración

Facilidad con la que el software interacciona con otros elementos de su entorno

Interoperabilidad

Capacidad de poder evaluar la correctitud de un sistema software

Testabilidad

Facilidad del software para que puedan ser corregidos errores detectados en él

Robustez

Capacidad del sistema para usar los recursos disponibles y el impacto que esto tiene sobre los tiempos de respuesta, rendimiento y consumo de memoria

Eficiencia

Capacidad de proteger un sistema de usos incorrectos, y, en el caso de que sea necesario pararlo, hacerlo de forma definida

Mantenibilidad

Respuesta parcialmente correcta.
Ha seleccionado correctamente 4.
La respuesta correcta es:

Facilidad con la que se pueden mover elementos entre distintos módulos o subsistemas, lo cual implica también cambios en los clientes de los mismos
→ Reestructuración,

Facilidad con la que el software interacciona con otros elementos de su entorno
→ Interoperabilidad,
Capacidad de poder evaluar la correctitud de un sistema software → Testabilidad,

Facilidad del software para que puedan ser corregidos errores detectados en él
→ Mantenibilidad,

Capacidad del sistema para usar los recursos disponibles y el impacto que esto tiene sobre los tiempos de respuesta, rendimiento y consumo de memoria
→ Eficiencia,

Capacidad de proteger un sistema de usos incorrectos, y, en el caso de que sea necesario pararlo, hacerlo de forma definida
→ Robustez

Pregunta 3
Correcta
Se puntuó 1,00 sobre 1,00
 Marcar pregunta

¿Qué es un diagrama de "cuadros y líneas" o de "bloques", usados, por ejemplo, en la propuesta de Rozansky?

a. Un diagrama para describir la perspectiva de seguridad
 b. Un diagrama para describir el sistema desde el punto de vista conceptual
 c. Un diagrama para describir el sistema desde cualquier punto de vista, usando una notación muy precisa en desarrollo de software para que pueda ser entendido por los ingenieros software e incluso por sistemas automáticos de desarrollo basado en modelos
 d. Un diagrama para describir el sistema desde cualquier punto de vista, usando una notación genérica y poco precisa pero que todas las partes interesadas puedan entender ✓

Respuesta correcta
La respuesta correcta es:
Un diagrama para describir el sistema desde cualquier punto de vista, usando una notación genérica y poco precisa pero que todas las partes interesadas puedan entender

Pregunta 4
Parcialmente correcta
Se puntuó 1,50 sobre 2,00
 Marcar pregunta

Elige entre los requisitos siguientes aquellos considerados perspectivas muy importantes en proyectos de gran envergadura, según Rozansky.

a. Seguro ✓
 b. Fácil de usar
 c. Accesible ✓
 d. Internacionalizable ✓
 e. Sostenible ✗

Respuesta parcialmente correcta.
Ha seleccionado correctamente 3.
Las respuestas correctas son:
Accesible,
Fácil de usar,
Internacionalizable,
Seguro

Pregunta 5
Correcta
Se puntuó 1,00 sobre 1,00
 Marcar pregunta

Indica la relación de "herencia" entre distintos principios de diseño arquitectónico

a. La modularización es un subtipo del principio de diseño arquitectónico de "separación de intereses"
 b. La abstracción es un ejemplo del principio de diseño arquitectónico de modularización
 c. El bajo acoplamiento y la alta cohesión son subtipos del principio de encapsulación
 d. La ocultación de la información y la separación entre interfaz e implementación son formas de llevar a cabo el principio de encapsulación ✓

Respuesta correcta
La respuesta correcta es:
La ocultación de la información y la separación entre interfaz e implementación son formas de llevar a cabo el principio de encapsulación

Pregunta 6
Correcta
Se puntuá 1,00 sobre 1,00

Ordena los pasos para describir un punto de vista

- Inquietudes
- Modelos
- Problemas y errores comunes
- Lista de comprobación

Respuesta correcta

Pregunta 7
Correcta
Se puntuá 1,00 sobre 1,00

El estándar IEEE P1471 considera puntos de vista pero no perspectivas en la Descripción Arquitectónica

Seleccione una:

- Verdadero ✓
- Falso

La respuesta correcta es 'Verdadero'

Pregunta 8
Correcta
Se puntuá 1,00 sobre 1,00

Las perspectivas arquitectónicas describen la arquitectura del software que será desarrollado de forma que se garantice el cumplimiento de los requisitos funcionales.

Seleccione una:

- Verdadero
- Falso ✓

La respuesta correcta es 'Falso'

Pregunta 9
Correcta
Se puntuá 1,00 sobre 1,00

P1471 no propone puntos de vista específicos, sólo que los que se elijan se entiendan y documenten bien

Seleccione una:

- Verdadero ✓
- Falso

La respuesta correcta es 'Verdadero'

Pregunta 10
Incorrecta
Se puntuá 0,00 sobre 1,00

Los puntos de vista son las reglas por las que se rigen las vistas concretas

Seleccione una:

- Verdadero
- Falso ✕

La respuesta correcta es "Verdadero"



Invita a otros estudiantes, crea contenido y gana los premios que te alegrarán el verano

Jnabre58

Cuestionario 4

<p>Pregunta 1 Correcta Se puntuá 2,00 sobre 2,00 <input type="checkbox"/> Marcar pregunta</p>	<p>Ordena las pruebas según el momento en el que se realiza, empezando por las que se realizan primero:</p> <p>4 ✓ Pruebas de sistema 1 ✓ Pruebas de unidad 6 ✓ Pruebas beta 3 ✓ Pruebas de integración 5 ✓ Pruebas de aceptación 2 ✓ Pruebas de componentes</p>
<p>Pregunta 2 Correcta Se puntuá 2,00 sobre 2,00 <input type="checkbox"/> Marcar pregunta</p>	<p>La verificación ✓ es el proceso de evaluar si el software cumple con los requisitos especificados al inicio.</p> <p>La validación ✓ es el proceso de evaluar si el software cumple de forma completa y correcta con su propósito original, es decir, hasta qué punto el producto satisface la intencionalidad con la que fue concebido por el cliente, aun cuando no se hubiera reflejado bien en las especificaciones.</p> <p>La calificación ✓ es el proceso de evaluar si el software cumple de forma completa y correcta con todos sus aspectos operativos que faciliten el mantenimiento.</p>
<p>Pregunta 3 Correcta Se puntuá 2,00 sobre 2,00 <input type="checkbox"/> Marcar pregunta</p>	<p>Escoge el concepto relacionado con las pruebas software que está siendo descrito</p> <p>Colección de software utilizado para la realización de pruebas software, especialmente las que se realizan de forma automática <input checked="" type="checkbox"/> testware</p> <p>Pruebas que tienen el cometido de descubrir errores asociados a la interfaz o interacción entre componentes <input checked="" type="checkbox"/> pruebas de integración</p> <p>Colección de software y datos de prueba usados por desarrolladores para las pruebas de unidad <input checked="" type="checkbox"/> arnés de prueba</p> <p>Los controladores ficticios (mocks) para pruebas de caja negra y los objetos ficticios (stubs) para hacer pruebas de unidad, son dos subtipos <input checked="" type="checkbox"/> dobles de prueba</p>
<p>Pregunta 4 Correcta Se puntuá 2,00 sobre 2,00 <input type="checkbox"/> Marcar pregunta</p>	<p>Elije de cada descripción si se refiere a garantía o a control de calidad</p> <p>Es competencia de todo el personal <input checked="" type="checkbox"/> Garantía de calidad</p> <p>Se concentra en el proceso de producción de un producto <input checked="" type="checkbox"/> Garantía de calidad</p> <p>Se concentra en el producto en sí <input checked="" type="checkbox"/> Control de calidad</p> <p>Es una tarea atribuida a un equipo dedicado a ella <input checked="" type="checkbox"/> Control de calidad</p>
<p>Pregunta 5 Correcta Se puntuá 2,00 sobre 2,00 <input type="checkbox"/> Marcar pregunta</p>	<p>Elije las pruebas que son consideradas dentro del factor operativo, según McCall</p> <p><input checked="" type="checkbox"/> a. Pruebas de estrés✓ <input type="checkbox"/> b. Pruebas de portabilidad <input type="checkbox"/> c. Pruebas de facilidad de testeo <input checked="" type="checkbox"/> d. Pruebas de correctitud✓ <input checked="" type="checkbox"/> e. Pruebas de seguridad✓ <input checked="" type="checkbox"/> f. Pruebas de fiabilidad✓</p>

Desarrollo de Software 22/23

WUOLAH

Hasta el
15/06/2023

Cuestionario 5

Pregunta 1 Correcta Se puntuá 1,00 sobre 1,00 F Marcar pregunta	Asocia las distintas clasificaciones de los lenguajes de modelado con el criterio de clasificación utilizado <p>Según nivel de abstracción: Computation Independent (CIM), platform independent (PIM) y platform specific (PSM) ✓</p> <p>Según dominio de aplicación: De uso general y de uso específico ✓</p> <p>Según punto de vista funcional: Estático y dinámico ✓</p>
Pregunta 2 Correcta Se puntuá 1,00 sobre 1,00 F Marcar pregunta	Asocia los distintos enfoques MDE con el nivel de abstracción al que pertenecen <p>Nivel bajo (enfoques concretos): Factoría software Web Service y Sistemas Interativos eXtensibles para móviles (XIS-Mobile) ✓</p> <p>Nivel alto: MDA, MDD, MDT ✓</p> <p>Nivel medio: Factorías software de Microsoft y entorno de trabajo de modelado de Eclipse ✓</p>
Pregunta 3 Parcialmente correcta Se puntuá 0,75 sobre 1,00 F Marcar pregunta	Selecciona entre los siguientes apartados, aquéllos que describan ejemplos de MDE (Model Driven Architecture) <p><input type="checkbox"/> a. Contenedores software para despliegue de aplicaciones (DOCKER)</p> <p><input checked="" type="checkbox"/> b. Model Driven Architecture (MDA) del OMG ✓</p> <p><input type="checkbox"/> c. Eclipse Modeling Framework</p> <p><input checked="" type="checkbox"/> d. Model Driven Testing ✓</p> <p><input checked="" type="checkbox"/> e. Microsoft Software Factories ✓</p>
Respuesta parcialmente correcta. Ha seleccionado correctamente 3. Las respuestas correctas son: Model Driven Architecture (MDA) del OMG, Microsoft Software Factories, Model Driven Testing, Eclipse Modeling Framework	
Pregunta 4 Correcta Se puntuá 1,00 sobre 1,00 F Marcar pregunta	UML es un ejemplo de meta-metamodelo, propio de la última capa (M3) del estándar Meta Object Facility (MOF) de MDA Seleccione una: <input type="radio"/> Verdadero <input checked="" type="radio"/> Falso ✓
Pregunta 5 Correcta Se puntuá 1,00 sobre 1,00 F Marcar pregunta	El compilador de Lisp está escrito en C Seleccione una: <input type="radio"/> Verdadero <input checked="" type="radio"/> Falso ✓
Pregunta 6 Correcta Se puntuá 1,00 sobre 1,00 F Marcar pregunta	La metaclass de una clase Ruby es un ejemplo de metamodelo Seleccione una: <input checked="" type="radio"/> Verdadero ✓ <input type="radio"/> Falso

Pregunta 7 Correcta Se puntuá 1,00 sobre 1,00 <input type="checkbox"/> Marcar pregunta	<p>Los marcadores que ayudan a mantener el software desarrollado con metodologías ágiles no evitan la necesidad de producir documentación de alto nivel</p> <p>Seleccione una:</p> <p><input checked="" type="radio"/> Verdadero ✓</p> <p><input type="radio"/> Falso</p>								
Pregunta 8 Correcta Se puntuá 1,00 sobre 1,00 <input type="checkbox"/> Marcar pregunta	<p>Asocia cada definición con el término que la define</p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%;">Sistema</td> <td>Conjunto ordenado de normas y procedimientos que regulan el funcionamiento de un grupo o colectivo</td> </tr> <tr> <td>Modelo</td> <td>Abstracción de un sistema bajo estudio</td> </tr> <tr> <td>Metamodelo</td> <td>Modelo que define la estructura de un lenguaje de modelado</td> </tr> <tr> <td>Meta-metamodelo</td> <td>Modelo que describe el lenguaje de modelado de un metamodelo</td> </tr> </table>	Sistema	Conjunto ordenado de normas y procedimientos que regulan el funcionamiento de un grupo o colectivo	Modelo	Abstracción de un sistema bajo estudio	Metamodelo	Modelo que define la estructura de un lenguaje de modelado	Meta-metamodelo	Modelo que describe el lenguaje de modelado de un metamodelo
Sistema	Conjunto ordenado de normas y procedimientos que regulan el funcionamiento de un grupo o colectivo								
Modelo	Abstracción de un sistema bajo estudio								
Metamodelo	Modelo que define la estructura de un lenguaje de modelado								
Meta-metamodelo	Modelo que describe el lenguaje de modelado de un metamodelo								
Pregunta 9 Correcta Se puntuá 1,00 sobre 1,00 <input type="checkbox"/> Marcar pregunta	<p>RoR tiene una herramienta de transformación (puente) PSM-PSM implícita que permite definir entidades como objetos Ruby y como tablas en un SGBD de forma simultánea y automática</p> <p>Seleccione una:</p> <p><input checked="" type="radio"/> Verdadero ✓</p> <p><input type="radio"/> Falso</p>								
Pregunta 10 Correcta Se puntuá 1,00 sobre 1,00 <input type="checkbox"/> Marcar pregunta	<p>El enfoque MD permite ser completamente productivos desde las primeras fases del ciclo de vida</p> <p>Seleccione una:</p> <p><input checked="" type="radio"/> Verdadero ✓</p> <p><input type="radio"/> Falso</p>								

Comenzado el lunes, 5 de junio de 2023, 22:22

Estado Finalizado

Finalizado en lunes, 5 de junio de 2023, 22:33

Tiempo empleado 11 minutos 15 segundos

Puntos 9,00/11,00

Calificación **8,18** de 10,00 (**81,82%**)

Pregunta 1

Correcta

Se puntuó 1,00 sobre 1,00

Ingredientes básicos del estándar IEEE P171 de descripción arquitectónica del software. Completa el texto:

Conjunto de normas de **[definición de términos]** ✓ tales como descripción arquitectónica, vista arquitectónica o punto de vista arquitectónico

Marco conceptual ✓ que sitúa estos términos en el contexto de los múltiples usos posibles de DAs para la construcción, análisis y evolución de sistemas

Conjunto de **[requerimientos]** ✓ sobre una DA de un sistema

El estándar IEEE P171 es **[independiente]** ✓ de la notación

Respuesta correcta

La respuesta correcta es:

Ingredientes básicos del estándar IEEE P171 de descripción arquitectónica del software. Completa el texto:

Conjunto de normas de **[definición de términos]** tales como descripción arquitectónica, vista arquitectónica o punto de vista arquitectónico

[Marco conceptual] que sitúa estos términos en el contexto de los múltiples usos posibles de DAs para la construcción, análisis y evolución de sistemas

Conjunto de **[requerimientos]** sobre una DA de un sistema

El estándar IEEE P171 es **[independiente]** de la notación



Pregunta 2

Correcta

Se puntuá 1,00 sobre 1,00

Relaciona las distintas propiedades no funcionales de la arquitectura software con su definición

Capacidad del sistema para usar los recursos disponibles y el impacto que esto tiene sobre los tiempos de respuesta, rendimiento y consumo de memoria

Eficiencia



Capacidad de proteger un sistema de usos incorrectos, y, en el caso de que sea necesario pararlo, hacerlo de forma definida

Robustez



Capacidad de poder evaluar la correctitud de un sistema software

Testabilidad



Facilidad del software para que puedan ser corregidos errores detectados en él

Mantenibilidad



Facilidad con la que el software interacciona con otros elementos de su entorno

Interoperabilidad



Facilidad con la que se pueden mover elementos entre distintos módulos o subsistemas, lo cual implica también cambios en los clientes de los mismos

Reestructuración



Respuesta correcta

La respuesta correcta es:

Capacidad del sistema para usar los recursos disponibles y el impacto que esto tiene sobre los tiempos de respuesta, rendimiento y consumo de memoria

→ Eficiencia,

Capacidad de proteger un sistema de usos incorrectos, y, en el caso de que sea necesario pararlo, hacerlo de forma definida

→ Robustez,

Capacidad de poder evaluar la correctitud de un sistema software → Testabilidad,

Facilidad del software para que puedan ser corregidos errores detectados en él

→ Mantenibilidad,

Facilidad con la que el software interacciona con otros elementos de su entorno

→ Interoperabilidad,

Facilidad con la que se pueden mover elementos entre distintos módulos o subsistemas, lo cual implica también cambios en los clientes de los mismos

→ Reestructuración



Pregunta 3

Correcta

Se puntuá 1,00 sobre 1,00

¿Qué es un diagrama de "cuadros y líneas" o de "bloques", usados, por ejemplo, en la propuesta de Rozansky?

- a. Un diagrama para describir el sistema desde cualquier punto de vista, usando una notación genérica y poco precisa pero que todas las partes interesadas puedan entender ✓
- b. Un diagrama para describir la perspectiva de seguridad
- c. Un diagrama para describir el sistema desde el punto de vista conceptual
- d. Un diagrama para describir el sistema desde cualquier punto de vista, usando una notación muy precisa en desarrollo de software para que pueda ser entendido por los ingenieros software e incluso por sistemas automáticos de desarrollo basado en modelos

Respuesta correcta

La respuesta correcta es:

Un diagrama para describir el sistema desde cualquier punto de vista, usando una notación genérica y poco precisa pero que todas las partes interesadas puedan entender

Pregunta 4

Correcta

Se puntuá 2,00 sobre 2,00

Elige entre los requisitos siguientes aquellos considerados perspectivas muy importantes en proyectos de gran envergadura, según Rozansky.

- a. Sostenible
- b. Internacionizable✓
- c. Fácil de usar✓
- d. Accesible✓
- e. Seguro✓

Respuesta correcta

Las respuestas correctas son:

Accesible,

Fácil de usar,

Internacionizable,

Seguro



Pregunta 5

Correcta

Se puntuá 1,00 sobre 1,00

Indica la relación de "herencia" entre distintos principios de diseño arquitectónico

- a. La ocultación de la información y la separación entre interfaz e implementación son formas de llevar a cabo el principio de encapsulación ✓
- b. El bajo acoplamiento y la alta cohesión son subtipos del principio de encapsulación
- c. La modularización es un subtipo del principio de diseño arquitectónico de "separación de intereses"
- d. La abstracción es un ejemplo del principio de diseño arquitectónico de modularización

Respuesta correcta

La respuesta correcta es:

La ocultación de la información y la separación entre interfaz e implementación son formas de llevar a cabo el principio de encapsulación

Pregunta 6

Correcta

Se puntuá 1,00 sobre 1,00

Ordena los pasos para describir un punto de vista

- ✓ Inquietudes
- ✓ Modelos
- ✓ Problemas y errores comunes
- ✓ Lista de comprobación

Respuesta correcta

Pregunta 7

Incorrecta

Se puntuá 0,00 sobre 1,00

El estándar IEEE P1471 considera puntos de vista pero no perspectivas en la Descripción Arquitectónica

Seleccione una:

- Verdadero
- Falso ✗

La respuesta correcta es 'Verdadero'



Pregunta **8**

Incorrecta

Se puntuá 0,00 sobre 1,00

Las perspectivas arquitectónicas describen la arquitectura del software que será desarrollado de forma que se garantice el cumplimiento de los requisitos funcionales.

Seleccione una:

- Verdadero ✗
 Falso

La respuesta correcta es 'Falso'

Pregunta **9**

Correcta

Se puntuá 1,00 sobre 1,00

P1471 no propone puntos de vista específicos, sólo que los que se elijan se entiendan y documenten bien

Seleccione una:

- Verdadero ✓
 Falso

◀ Cuestionario tema 2

Ir a...

Taller RoR3: usuarios y sesiones ►

Pregunta **10**

Correcta

Se puntuá 1,00 sobre 1,00

Los puntos de vista son las reglas por las que se rigen las vistas concretas

Seleccione una:

- Verdadero ✓
 Falso

La respuesta correcta es 'Verdadero'



Comenzado el martes, 6 de junio de 2023, 08:55

Estado Finalizado

Finalizado en martes, 6 de junio de 2023, 09:01

Tiempo empleado 5 minutos 35 segundos

Puntos 11,00/11,00

Calificación 10,00 de 10,00 (100%)

Pregunta 1

Correcta

Se puntuó 1,00 sobre 1,00

Ingredientes básicos del estándar IEEE P171 de descripción arquitectónica del software. Completa el texto:

Conjunto de normas de [definición de términos] ✓ tales como descripción arquitectónica, vista arquitectónica o punto de vista arquitectónico

[Marco conceptual] ✓ que sitúa estos términos en el contexto de los múltiples usos posibles de DAs para la construcción, análisis y evolución de sistemas

Conjunto de [requerimientos] ✓ sobre una DA de un sistema

El estándar IEEE P171 es [independiente] ✓ de la notación

Respuesta correcta

La respuesta correcta es:

Ingredientes básicos del estándar IEEE P171 de descripción arquitectónica del software. Completa el texto:

Conjunto de normas de [definición de términos] tales como descripción arquitectónica, vista arquitectónica o punto de vista arquitectónico

[Marco conceptual] que sitúa estos términos en el contexto de los múltiples usos posibles de DAs para la construcción, análisis y evolución de sistemas

Conjunto de [requerimientos] sobre una DA de un sistema

El estándar IEEE P171 es [independiente] de la notación



Pregunta 2

Correcta

Se puntuó 1,00 sobre 1,00

Relaciona las distintas propiedades no funcionales de la arquitectura software con su definición

Facilidad con la que el software interacciona con otros elementos de su entorno

Interoperabilidad



Facilidad del software para que puedan ser corregidos errores detectados en él

Mantenibilidad



Facilidad con la que se pueden mover elementos entre distintos módulos o subsistemas, lo cual implica también cambios en los clientes de los mismos

Reestructuración



Capacidad de poder evaluar la correctitud de un sistema software

Testabilidad



Capacidad de proteger un sistema de usos incorrectos, y, en el caso de que sea necesario pararlo, hacerlo de forma definida

Robustez



Capacidad del sistema para usar los recursos disponibles y el impacto que esto tiene sobre los tiempos de respuesta, rendimiento y consumo de memoria

Eficiencia



Respuesta correcta

La respuesta correcta es:

Facilidad con la que el software interacciona con otros elementos de su entorno

→ Interoperabilidad,

Facilidad del software para que puedan ser corregidos errores detectados en él

→ Mantenibilidad,

Facilidad con la que se pueden mover elementos entre distintos módulos o subsistemas, lo cual implica también cambios en los clientes de los mismos

→ Reestructuración,

Capacidad de poder evaluar la correctitud de un sistema software → Testabilidad,

Capacidad de proteger un sistema de usos incorrectos, y, en el caso de que sea necesario pararlo, hacerlo de forma definida

→ Robustez,

Capacidad del sistema para usar los recursos disponibles y el impacto que esto tiene sobre los tiempos de respuesta, rendimiento y consumo de memoria

→ Eficiencia



Pregunta 3

Correcta

Se puntuá 1,00 sobre 1,00

¿Qué es un diagrama de "cuadros y líneas" o de "bloques", usados, por ejemplo, en la propuesta de Rozansky?

- a. Un diagrama para describir el sistema desde cualquier punto de vista, usando una notación genérica y poco precisa pero que todas las partes interesadas puedan entender ✓
- b. Un diagrama para describir el sistema desde cualquier punto de vista, usando una notación muy precisa en desarrollo de software para que pueda ser entendido por los ingenieros software e incluso por sistemas automáticos de desarrollo basado en modelos
- c. Un diagrama para describir la perspectiva de seguridad
- d. Un diagrama para describir el sistema desde el punto de vista conceptual

Respuesta correcta

La respuesta correcta es:

Un diagrama para describir el sistema desde cualquier punto de vista, usando una notación genérica y poco precisa pero que todas las partes interesadas puedan entender

Pregunta 4

Correcta

Se puntuá 2,00 sobre 2,00

Elige entre los requisitos siguientes aquellos considerados perspectivas muy importantes en proyectos de gran envergadura, según Rozansky.

- a. Internacionalizable✓
- b. Fácil de usar✓
- c. Seguro✓
- d. Accesible✓
- e. Sostenible

Respuesta correcta

Las respuestas correctas son:

Accesible,

Fácil de usar,

Internacionalizable,

Seguro



Pregunta 5

Correcta

Se puntuá 1,00 sobre 1,00

Indica la relación de "herencia" entre distintos principios de diseño arquitectónico

- a. La modularización es un subtipo del principio de diseño arquitectónico de "separación de intereses"
- b. La ocultación de la información y la separación entre interfaz e implementación son formas de llevar a cabo el principio de encapsulación ✓
- c. La abstracción es un ejemplo del principio de diseño arquitectónico de modularización
- d. El bajo acoplamiento y la alta cohesión son subtipos del principio de encapsulación

Respuesta correcta

La respuesta correcta es:

La ocultación de la información y la separación entre interfaz e implementación son formas de llevar a cabo el principio de encapsulación

Pregunta 6

Correcta

Se puntuá 1,00 sobre 1,00

Ordena los pasos para describir un punto de vista

- ✓ Inquietudes
- ✓ Modelos
- ✓ Problemas y errores comunes
- ✓ Lista de comprobación

Respuesta correcta

Pregunta 7

Correcta

Se puntuá 1,00 sobre 1,00

El estándar IEEE P1471 considera puntos de vista pero no perspectivas en la Descripción Arquitectónica

Seleccione una:

- Verdadero ✓
- Falso

La respuesta correcta es 'Verdadero'



Pregunta 8

Correcta

Se puntuá 1,00 sobre 1,00

Las perspectivas arquitectónicas describen la arquitectura del software que será desarrollado de forma que se garantice el cumplimiento de los requisitos funcionales.

Seleccione una:

- Verdadero
- Falso ✓

La respuesta correcta es 'Falso'

Pregunta 9

Correcta

Se puntuá 1,00 sobre 1,00

P1471 no propone puntos de vista específicos, sólo que los que se elijan se entiendan y documenten bien

Seleccione una:

- Verdadero ✓

◀ Cuestionario tema 2

Ir a...

Taller RoR3: usuarios y sesiones ►

Pregunta 10

Correcta

Se puntuá 1,00 sobre 1,00

Los puntos de vista son las reglas por las que se rigen las vistas concretas

Seleccione una:

- Verdadero ✓
- Falso

La respuesta correcta es 'Verdadero'



examen-2015-resultados.pdf



Anónimo



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

**ENGINEERING
THE
FUTURE**

¿Crees en el poder de la tecnología
para cambiar el mundo?

¡Indra te espera en Twitch!

La defensa, la movilidad y el sector
aeroespacial necesitan tu talento



7 de junio - 17 hs
¡Inscríbete!

indra

**La mejor posición es siempre
encima (de una de estas).**



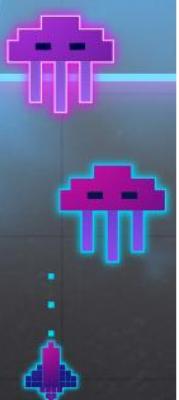
la quiero

**Newskill
Neith Zephyr**



NEWSKILL

Lo bueno de
estas sillas es
que se reclinan.
Los usos que le
puedas dar te
los imaginas tú.



Nuestra silla gaming
Neith Zephyr se
adapta a todas las
circunstancias, desde
intensas jornadas de
estudio a largas
jornadas de juego. Su
acabado en tela
transpirable es
perfecto para verano.
Disponible en ocho
colores.

newskillgaming.com

1

Examen final

Nombre:

22-06-2015

I- Cada respuesta incorrecta restará 1/2 de la puntuación obtenida por cada respuesta correcta.

1. **(0,5)** Seleccionar la única respuesta incorrecta a la siguiente cuestión sobre el concepto de *patrón de diseño* en el desarrollo de software.
 - (a) Sirve para resolver un problema concreto en un determinado contexto.
 - (b) La factibilidad de la solución que se propone ha de estar comprobada.
 - (c) ✓ Un patrón de diseño no se cambia, sino que se adapta mediante extensión al sistema-software concreto.
 - (d) Generan indirectamente la solución al problema que se proponen resolver.
2. **(0,5)** Seleccionar la única respuesta correcta a la siguiente cuestión sobre el patrón de diseño *Abstracción / Caso*. Suponer que se va a crear un catálogo de piezas de coche:
 - (a) Crear cada pieza disponible como una instancia de subclase de Pieza ={Nombre, Fabricante, Precio, Código-Almacén, Fecha-Entrada, Disponibles } con su código de barras.
 - (b) Crear clases-Pieza específicas y diferentes por cada fabricante, que a su vez son subclases de la clase Pieza.
 - (c) Crear una clase Pieza como una agregación de todas las piezas disponibles, que tienen como atributo sólo su código de barras.
 - (d) ✓ Mediante una asociación, la clase Pieza se materializa en las piezas de ese tipo que quedan en el almacén.
3. **(0,5)** Seleccionar la única respuesta correcta a la siguiente cuestión sobre el patrón de diseño *Jerarquía General*. Suponer que se va a crear un catálogo de piezas de coche que pueden tener partes. La utilización más correcta del patrón anterior supone que:
 - (a) Para representar la jerarquía de piezas del catálogo sólo necesitaremos utilizar subclasi-ficación (herencia entre clases de piezas).
 - (b) Se ha definir una Pieza-compuesta y una Pieza-simple, ambas como agregaciones de la clase Pieza.
 - (c) ✓ Se establece sólo una asociación entre Pieza y Pieza-compuesta.
 - (d) Se establecen asociaciones entre Pieza, Pieza-compuesta y Pieza-simple
4. **(0,5)** Seleccionar la única respuesta incorrecta a la siguiente cuestión sobre el patrón de diseño *Inmutable*.
 - (a) Los valores de las variables de instancia de los objetos sólo se pueden asignar o modificar en el método constructor de la clase.
 - (b) Un método que necesite modificar el estado de la instancia ha de llamar necesariamente al método constructor.
 - (c) ✓ Los objetos inmutables no pueden tener variables públicas.
 - (d) La ejecución de cualquier método de instancia no puede cambiar el valor de las variables propias del objeto.

WUOLAH

Nota: Mutable Objects/Clases don't provide "setter" methods — methods that modify fields or objects referred to by fields
 -Make all fields final and private
 -Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
 -If the instance fields include references to mutable objects, don't allow those objects to be changed. (<https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html>)

5. (0,5) Seleccionar la única respuesta correcta a la siguiente cuestión sobre el patrón de diseño *Factoría Abstracta*.
- La ejecución del único método del marco de trabajo devuelve a la entidad demandante de creación de instancias una referencia de objeto.
 - Factoría* ha de ser siempre una clase abstracta.
 - El método *crearInstancia()* de las factorías siempre devuelve lo mismo.
 - La clase de objetos específica del marco de trabajo implementa la interfaz *ClaseGenerica*
6. (0,5) Seleccionar la respuesta incorrecta a la siguiente cuestión sobre el patrón de diseño *Visitante*.
- Los visitantes concretos guardan el estado durante el recorrido de la estructura de objetos.
 - A la operación *visitar()* de un visitante concreto hay que pasarle la referencia de objeto del elemento que ha llamado al método *aceptar(...)*.
 - Elemento* no posee subclases.
 - Los objetos atienden llamadas a su operación *aceptar()* pero no pueden resolver por sí solos cuál es el visitante concreto que ha de actuar a continuación.

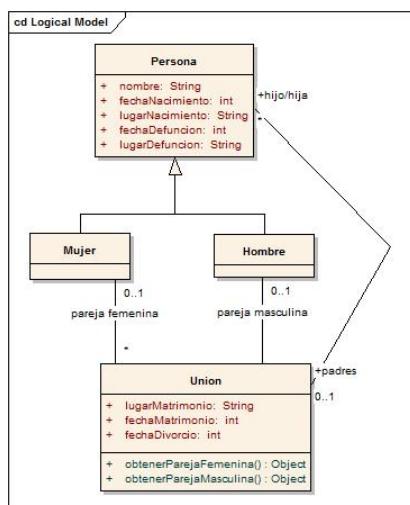


Figura 1: Árbol genealógico familiar

II- Ejercicios sobre la teoría impartida

1. (0,5) Encontrar todas las situaciones en las que el patrón *Delegation* podría ser aplicado en el diagrama de clases de la figura 1 que representa un árbol genealógico.

Solución: `obtenerMadre() /obtenerPadre() : Persona;` la clase de los objetos que se devuelven programa estos métodos utilizando el mecanismo de *delegación* –a través de la asociación *hijo-hija/padres*– para lo cual delega en los métodos `obtenerParejaFemenina() /Masculina()` de la clase *Union*.

**ENGINEERING
THE
FUTURE**

¿Crees en el poder de la tecnología para cambiar el mundo?

¡Indra te espera en Twitch!

No faltés a la experiencia digital en la que conocerás las tecnologías, los proyectos y las personas que están transformando el futuro.

¡La defensa, la movilidad y el sector aeroespacial necesitan tu talento!
Asómate al futuro y haz ingeniería del mañana, hoy.



7 de junio - 17 hs
¡Inscríbete!



indra

2. (1,0) Calcular el número de McCabe o ciclomático por los tres métodos estudiados: (a) construyendo un grafo, (b) geometría del plano, (c) a partir del número de condicionales para el siguiente programa simple:

```

1 #define N 200
2 int main(){
3     int i , j;
4     int M[N][N];
5     bool v[N];
6     for (i=0; i<N; ++i)
7         for (j=0; j<N; ++j)
8             M[i][j]= i + j;
9     for (i=0; i<N; ++i)
10        if(M[i][j] %2 == 0)
11            v[i]= true;
12        else
13            v[i]= false;
14    return 0;
15 }
```

Solución:

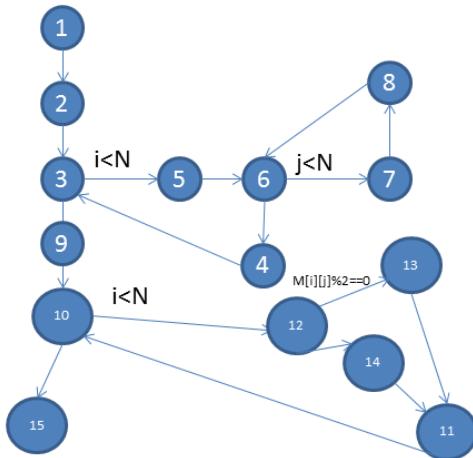


Figura 2: Grafo de flujo equivalente

- a aristas= 18; nodos= 15; N= 5
- b 5 regiones en el plano
- c condicionales= 4; N= 5

3. (1,0) Una clase *singleton* que se denomina *Quiz* ha de provocar la creación de *preguntas* (objetos) de varios tipos (sólo texto, con sonidos o imágenes) para un aplicación de móvil. La idea para implementar el juego consiste en que, dependiendo del tipo de pregunta que se ha generado en la actividad principal del juego, se crearán las instancias de *PreguntaTexto*, *PreguntaSonidos*, o *PreguntaImagenes* apropiada.

Utilizando el patrón de diseño factoría abstracta dibujar un diagrama de clases en el que han de aparecer, al menos, las siguientes clases: *ActividadPrincipal*, *Pregunta*, *PreguntaTexto*, *PreguntaSonidos*, *PreguntaImagenes*, *FactoriaPreguntas*, *FactoriaPreguntasTexto*, *FactoriaPreguntasSonidos*, y *FactoriaPreguntasImagenes*. Para que se considere correctamente realizado

Desarrollo de Software - Grupo A

4

hay que programar además la selección de la factoría correcta en la actividad principal dependiendo del tipo de pregunta. Los métodos de creación de las factorías concretas no devolverán el mismo tipo que el método de creación de *FactoriaPreguntas*.

Solución:

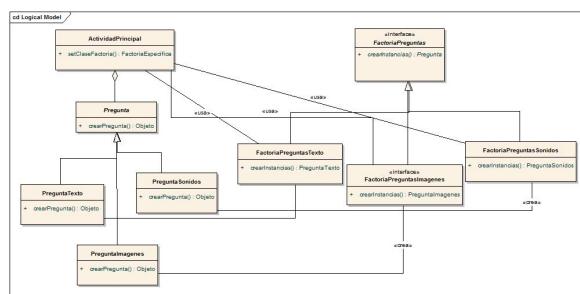


Figura 3: Diagrama de clases del patrón *Factoría abstracta*

4. **(1,0)** Un programador afirma que sus programas están libres de defectos con un nivel de certeza del 80 %. ¿ Con cuántos defectos debe sembrarse el programa que ha realizado antes de la prueba a fin de probar la afirmación anterior del programador? Nota: el plan de prueba establece en este caso que ha que continuar las pruebas del programa hasta encontrar la totalidad de los defectos sembrados.

$$\begin{aligned}
 C &= \frac{S}{S - N + 1} \\
 C &= 0,80 \\
 S &= \text{sembrados} \\
 N &= \text{reales} = 0
 \end{aligned}$$

Solución: S= 4 defectos

III- Preguntas cortas

1. **(0,5)** Indicar tres posibles usos de los patrones en el diseño orientado a objetos de un sistema software. **Solución:**
 - Modelado:** sirve para identificar y representar entidades del mundo real como objetos informáticos
 - Estructuración:** sirve para descomponer un objeto complejo en otros más simples
 - Interfaz:** sirve para determinar qué atributos no es necesario que sean visibles desde donde se vayan a utilizar esos objetos
2. **(1,0)** Justificar el patrón de diseño que se elegiría para diseñar un sistema software en el cual un conjunto de entidades (objetos en el diseño OO) comparten información pero se diferencian en aspectos importantes. Queremos evitar a toda costa replicar información común a las mencionadas entidades, prever inconsistencias de información y poder cambiar características particulares en los objetos. Representar la solución propuesta con un diagrama de clases UML. **Solución:** Pueden servir los patrones **Abstracción/Caso** o **Flyweight**, en este segundo caso, la mayor parte del estado de los objetos se puede convertir en **extrínseco** de tal forma que muchos de los objetos pueden ser reemplazados por unos pocos objetos compartidos.

3. (0,5) ¿Cuáles de los siguientes elementos no tiene una influencia importante en los costos de mantenimiento de un sistema software? Justificar la respuesta.
- Tipo de aplicación (transformacional, reactiva, tiempo–real, etc.)
 - Instalación de una nueva aplicación del sistema
 - Tipo de sistema (S, P o E)
 - Duración prevista del ciclo de mantenimiento y evolución
 - Actualización con el hardware más novedoso y de mayor rendimiento
 - Plataforma (sistema operativo y software de red) en que está instalado
 - Calidad de diseño del sistema
 - Calidad de la documentación

IV- Supuesto Práctico

- (1,5) Explicar brevemente la ecuación fundamental del modelo predictivo de costes de mantenimiento de Belady-Lehman $M = P + K \times c - d$ y contestar a la siguiente cuestión justificándola
- ¿ Cuál de las siguientes decisiones eliges como más costo–efectiva para el mantenimiento futuro de un sistema software, que acaba de ser entregado, que posee una documentación poco clara y alta complejidad ?($c = 48$, ver Tabla 1) :
 - Pedir una nueva documentación (10,000 EUR)
 - Devolver el software para que lo refactoricen y bajaran la complejidad ciclomática un 50 % (hora programador= 80 EUR, 10,000 líneas de código, 40 paquetes y 120 componentes)
 - Cambiar el sistema operativo, la constante K disminuye su valor el 20 %.

complejidad (c)	riesgo	tiempo prueba (horas)
1–10	bajo	10/1000 líneas
11–20	moderado	50/1000 líneas
21–50	alto	100/1000 líneas
51+	imposible tests	—

Tabla 1: Valoración de riesgos según el número ciclomático

Solución:

- Indica una predicción de los costes de mantenimiento en función de tres factores: P se refiere al esfuerzo necesario para desarrollar ese software; c a la complejidad ciclomática del producto software, que viene multiplicada por una constante de tipo empírico; d a la familiaridad que tienen los responsables de mantenimiento con el software en cuestión.

- b) La opción (i) aumentaría la familiaridad con el software de los miembros del equipo responsable y, por consiguiente, podría llegar a reducir considerablemente el coste de mantenimiento y no es muy cara considerando los beneficios potenciales; (ii) puede resultar muy cara ya que se puede observar claramente que se trata de una aplicación muy compleja y extensa, puede resultar muy caro reducir su complejidad en un 50 % y puede ocasionar replantearse incluso el diseño del software, además después de reducir la complejidad nada nos garantiza que la documentación siga siendo mala e insuficiente; (iii) es incluso menos efectiva que la opción (ii) ya que el término de la complejidad sólo se ve reducido en un 20 %, además el esfuerzo de mantenimiento y la falta de experiencia con ese software se mantendrán inalteradas.

examen-2014-septiembre-resultado...



Anónimo



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

**ENGINEERING
THE
FUTURE**

¿Crees en el poder de la tecnología
para cambiar el mundo?

¡Indra te espera en Twitch!

La defensa, la movilidad y el sector
aeroespacial necesitan tu talento



7 de junio - 17 hs
¡Inscríbete!

indra

Tu dosis diaria de Wall Street en solo 5 minutos.

Suscríbete



(y cómete a Wall Street)



1

Convocatoria extraordinaria de septiembre

Nombre:

5-09-2014

I- Cada respuesta incorrecta restará 1/2 de la puntuación obtenida por cada respuesta correcta.

1. **(0,5)** Seleccionar la única respuesta correcta a la siguiente cuestión sobre el concepto de *patrón de diseño* en el desarrollo de software:
 - (a) Las clases de un patrón de diseño se pueden modificar para adaptarse a las condiciones particulares de un problema que se repite.
 - (b) Un patrón de diseño es básicamente un esquema algorítmico (como “divide y vencerás”, “programación dinámica”, etc.) pero expresado con clases
 - (c) Cada patrón es una estructura de clases fija que se adapta al desarrollo de una determinada aplicación o sistema-software.
 - (d) Un patrón propone una solución a un tipo de problema para cualquier plataforma de ejecución.
2. **(0,5)** Seleccionar la única respuesta correcta a la siguiente cuestión sobre las diferencias entre patrones de diseño y marcos de trabajo (*frameworks*) en sistemas software:
 - (a) Los patrones de diseño son menos abstractos que los marcos de trabajo.
 - (b) Los marcos de trabajo son un conjunto de clases que se han de utilizar sin modificación.
 - (c) Un marco de trabajo es el esqueleto de una aplicación que ha de ser adaptado a las necesidades concretas por el programador de la aplicación.
 - (d) Un patrón de diseño puede estar compuesto por varios marcos de trabajo.
3. **(0,5)** Indicar para cada uno de los siguientes ejemplos de sistemas a qué tipo de sistemas (S, P o E) pertenece:
 - (a) Un sistema de reservas de vuelos y hoteles. **E**
 - (b) Un sistema de prevención sísmica basado en las lecturas de una red de sismógrafos. **P**
 - (c) Un programa para jugar al ajedrez. **P**
 - (d) Un sistema operativo. **P**
 - (e) Un sistema para control automático de vuelo de una aeronave. **P**
 - (f) Un sistema informático para modelar la Economía Mundial. **E**
4. **(0,5)** Seleccionar cuál de los elementos siguientes no tiene influencia importante en los costos de mantenimiento de un sistema software :
 - (a) Tipo de aplicación que hay que mantener (transformacional, reactiva, tiempo-real, etc.)
 - (b) Tipo de sistema software: S, P o E
 - (c) Duración prevista del ciclo de mantenimiento y evolución
 - (d) Sistema operativo y software de red en que está instalado el sistema
 - (e) Calidad de la documentación del software
 - (f) Diseño del sistema

 Zumitow.

que debes saber

Si mercados la semana ha ido bastante «difiel» ha sido una semana de incertidumbre. No queremos que pierdas que la situación seguirá inestable y no sabemos si la diferencia entre los mercados ya que se recuerdan bien reciente acuerdo en ambos frentes es histórica.

Stock de hoy

El ETF Favorito de Greta Thunberg



Hoy no vamos a evaluar un stock en un ETF, que tiene mucho jugando en su favor y que ha roto los \$35K, evolucionando Energía Limpia que invita al S&P 500.

WUOLAH

5. (0,5) De las siguientes afirmaciones relacionadas con la aplicación de MDA al desarrollo de software, seleccionar la única incorrecta:
- (a) De acuerdo con el modelo de desarrollo por transformación de modelos MDA, pueden existir desarrollos de un sistema software en los que el mismo tipo de diagrama (Clases, Estados-UML, Secuencia, Actividad, etc.) pertenece a distintos tipos de modelos intermedios (PIM, PSM, etc.)
 - (b) Un mismo modelo expresado en una notación determinada puede ser considerado un modelo MDA diferente dependiendo de la plataforma objetivo final.
 - (c) ✓ Si realizamos pruebas unitarias de software necesitamos tener desarrollados todos los componentes del sistema antes de comenzar a probarlo.
 - (d) No se puede, en general, obtener automáticamente un modelo PIM a partir de un modelo CIM de MDA.
 - (e) Las pruebas que hay que desarrollar para validar un sistema software son: unitarias, integración, validación y del sistema.
6. (0,5) Seleccionar la única alternativa correcta sobre evolución del software:
- (a) El mantenimiento del software posee un ciclo de proceso independiente del ciclo de desarrollo del software
 - (b) El modelo predictivo de Belady–Lehman expresa la relación entre el coste en desarrollar el sistema (p), la complejidad del software (c) y la buena documentación (d) del mismo, que se resume con la siguiente ecuación $M = p + K \times c - d$. La constante empírica K se calcula después de la instalación y configuración para una plataforma software concreta
 - (c) El factor (SU) de valoración de la comprensión del código para el correcto mantenimiento del software según el modelo COCOMO depende fundamentalmente de la alta cohesión y bajo acoplamiento entre componentes (módulos, clases, paquetes) del sistema
 - (d) ✓ Los mantenimientos perfectivo y correctivo del software *deterioran* a un sistema software
7. (0,5) ¿Cuáles de los siguientes pasos del proceso sistemático de prueba de una aplicación Web no pertenecen al protocolo explicado?:
- (a) Revisar el modelo de contenidos
 - (b) Comprobar el modelo de interfaz respecto de los casos de uso
 - (c) ✓ Detectar errores en el modelo de requisitos
 - (d) Ejercitarse la navegación del modelo de interfaz de usuario
 - (e) Detectar errores de navegación en el modelo de interfaz
 - (f) Pruebas unitarias de componentes Web
 - (g) Revisar la facilidad de navegación a través de toda la arquitectura software
 - (h) ✓ Buscar enlaces rotos antes de instalar las páginas
 - (i) Comprobar compatibilidad con plataformas y sus configuraciones
 - (j) Pruebas de seguridad, robustez, stress, carga y rendimiento

**ENGINEERING
THE
FUTURE**

¿Crees en el poder de la tecnología para cambiar el mundo?

¡Indra te espera en Twitch!

No faltes a la experiencia digital en la que conocerás las tecnologías, los proyectos y las personas que están transformando el futuro.

¡La defensa, la movilidad y el sector aeroespacial necesitan tu talento!
Asómate al futuro y haz ingeniería del mañana, hoy.



7 de junio - 17 hs
¡Inscríbete!



indra

II- Ejercicios sobre la teoría impartida

1. (2,5) Utilizar el patrón de diseño *factoría abstracta* para realizar el diagrama de clases de una aplicación que simula 2 carreras de bicicletas: montaña y carretera:

- Carrera es una interfaz de Java, que representa a la *factoría* del patrón, que declara los métodos abstractos de fabricación: `crearCuadro()`, `crearManillar()`, `crearRuedas()`.
- CarreraMontaña y CarreraCarretera son las clases factoría específicas o concretas que se utilizan para crear los objetos específicos: `cuadro_montaña`, `cuadro_carretera`, `manillar_montaña`, `manillar_carretera`, `ruedas_montaña` y `ruedas_carretera`
- La clase genérica del patrón que se pide se llamará Bicicleta (clase abstracta) e incluirá los métodos:
 - Método constructor: `+Bicicleta (TC b);`
 - `public enum TC{ MONTANA, CARRETERA}`
 - el método `-crearBicicleta()`, que se encarga de construir una bicicleta como una agregación de objetos de Cuadro, Manillar y Ruedas
 - Esta clase usa los métodos fabricación declarados en la interfaz factoría abstracta Carrera del patrón.
- Las clases específicas CuadroMontaña, ManillarMotaña y RuedasMontaña; y CuadroCarretera, ManillarCarretera y RuedasCarretera son todas ellas subclases de las clases genéricas: Manillar, Cuadro y Ruedas

Por último, el programa principal incluye las clases Montaña y Carretera que se encargan de obtener las bicicletas necesarias para montar una carrera de montaña o de carretera, respectivamente.

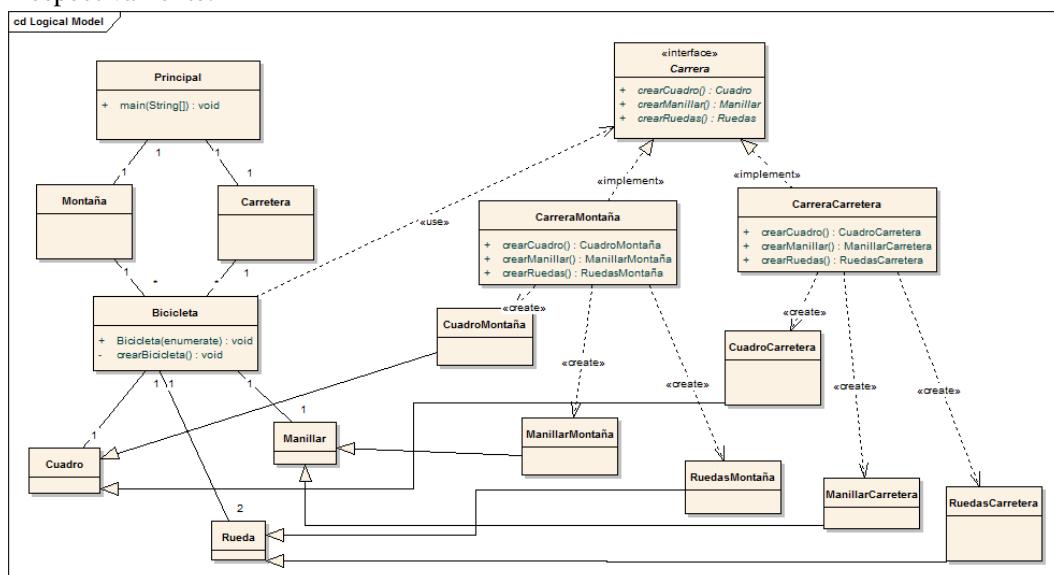


Figura 1: Diagrama de clases utilizando el patrón *factoría abstracta*

2. (1,5) Para el siguiente código, determinar su número ciclomático utilizando los tres métodos diferentes estudiados en clase: (a) construir el grafo de flujo y contar nodos y arcos; (b) obteniendo las sentencias condicionales y (c) regiones espaciales en que se divide al plano.

Tu dosis diaria de Wall Street en solo 5 minutos.

Suscribete



(y cómete a Wall Street)



Desarrollo de Software - Grupo A

4

```

1 public void drawScore(int n){
2     while(numdigitos >0){
3         score[numdigitos].erase();
4     }
5     numdigitos=0;
6     if(n==0){
7         //liberar memoria del objeto
8         delete(score[numdigitos]);
9         score[numdigitos]= new Displayable(digitos[0]);
10        score[numdigitos].move(Point((700-numdigitos*18),40));
11        score[numdigitos].draw();
12        numdigitos++;
13    }
14    while(n){
15        int resto= n % 10;
16        delete(score[numdigitos]);
17        score[numdigitos]= new Displayable(digitos[resto]);
18        score[numdigitos].move(Point((700-numdigitos*18),40));
19        score[numdigitos].draw();
20        n= n/10;
21        numdigitos++;
22    }
23 }
```

Solución:

- Número de nodos (N) = 6; Número de arcos (A) = 8; Número McCabe = A - N + 2 = 4
- 1 sentencia if y 2 whiles; Número McCabe = sentencias condicionales + 1 = 4
- El plano se divide en 4 subregiones (ver figura 2)

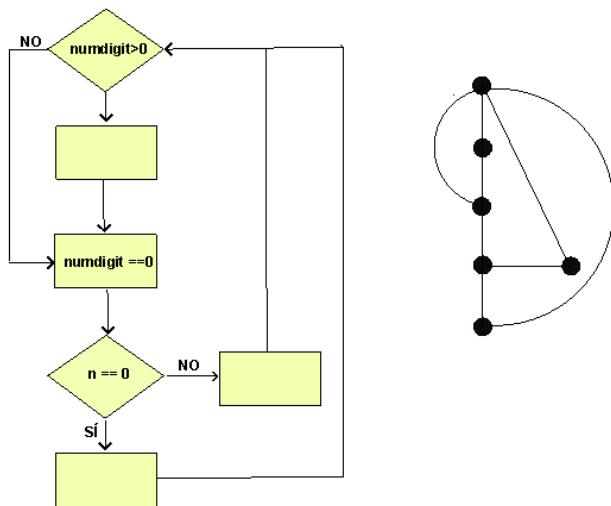


Figura 2: Grafo de flujo



3. (1,0) Una expresión de un lenguaje de programación se puede descomponer en una jerarquía de subexpresiones. Por ejemplo, $(a / (b+c))$ es una de las subexpresiones contenidas en la expresión terminal: $(a / (b+c)) + (b - \text{func}(d) * (e / (f+g)))$. Utilizando uno de los patrones Jerarquía General o Composite, crear un diagrama de clases para representar expresiones sintácticamente correctas para un lenguaje de programación en particular, por ejemplo, Java.

Solución:

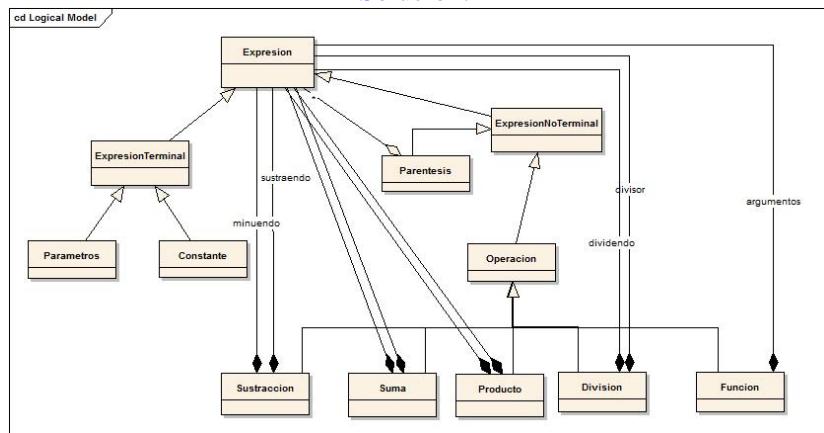


Figura 3: Diagrama de clases para expresiones Java correctas sintácticamente

IV- Supuesto Práctico

1. (1,5) Considerando el siguiente programa Java cuando se ejecuta: (a)sin argumentos, Salida:“Debes especificar un argumento”. En este caso se produce una excepción que es capturada en el programa principal `main(...)`, en la línea (101).
 (b) con ‘-’ como argumento Salida:“El valor numerico de -MiOtraExcepcion:la cadena solo contiene ‘-’ Manejado en el punto 2”. En este caso se produce una excepción en la línea (50) del método estático `c(String s, int base)` que es capturada en la línea (64) del método `b(String s, int base)`.
 y (c) con :

- ‘675’: Salida:“El valor numerico de 675 en base 10 es 675”. En este caso no se produce ninguna excepción, ya que se ejecutan las instrucciones(60–62) del método `b(String s, int base)`, que llama al `c(String s, int base)`; después de ejecutar el bloque (21–41) devolverá el valor de resultado tras asignar correctamente el signo,
- ‘-67’, Idem que el anterior. Después de cambiar el signo de resultado, en las líneas (60) y (62) se escribirá: “El valor numerico de -67 en base 10 es -67”
- ‘A’ Salida:“El valor numérico de A MiExcepcion:el carácter no es un dígito:’0’..’9’ Manejado en el punto 1”. En este caso se produce una excepción en la línea (30) del método estático `c(String s, int base)` que es capturada en la línea (79) del método `a(String s, int base)`, escribiéndose la parte final del mensaje con las instrucciones del bloque (83–85). y

- '7C8'. Se trata de un caso similar al anterior, cuando se lea el carácter 'C' de la hilera de entrada, se levantará MiExcepcion en la línea (38) del método estático c(String s, int base); continuando el resto de la ejecución como en el caso anterior.

Indicar la salida que producirá en cada uno de los casos (a)-(c) anteriores y explicar cómo funciona la gestión de las excepciones que se han implementado (indicar el orden de ejecución de las instrucciones del programa cuando se produzca cada una de las excepciones programadas).

```

1  class MiExcepcion extends Exception{
2      public MiExcepcion(){ super(); }
3      public MiExcepcion(String s){ super(s); }
4  }
5  class MiOtraExcepcion extends Exception{
6      public MiOtraExcepcion(){ super(); }
7      public MiOtraExcepcion(String s){ super(s); }
8  }
9  class MiSubExcepcion extends MiExcepcion{
10     public MiSubExcepcion(){ super(); }
11     public MiSubExcepcion(String s){ super(s); }
12 }
13
14 public class ParseIntegerDemo {
15     public static int c(String s, int base) throws MiExcepcion, MiOtraExcepcion{
16         int resultado = 0;
17         boolean negativo = false;
18         int i = 0;
19         int digito;
20         int max=s.length();
21
22         if (max > 0) {
23             if (s.charAt(0) == '-') {
24                 negativo = true;
25                 i++;
26             }
27             if (i < max) {
28                 digito = Character.digit(s.charAt(i++),base);
29                 if (digito < 0) {
30                     throw new MiExcepcion("el_caracter_no_es_un_digito:'0'..'9'");
31                 } else {
32                     resultado = -digito;
33                 }
34             }
35             while (i < max) {
36                 digito = Character.digit(s.charAt(i++),base);
37                 if (digito < 0) {
38                     throw new MiExcepcion("el_caracter_no_es_un_digito:'0'..'9'");
39                 }
40                 resultado *= base;
41                 resultado -= digito;
42             }
43         } else {
44             throw new MiSubExcepcion("la_cadena_de_entrada's'_esta_vacia");
45         }
46         if (negativo) {
47             if (i > 1) {
48                 return resultado;
49             } else {
50                 throw new MiOtraExcepcion("la_cadena_solo_contiene '-'");
51             }
52         } else {
53             return -resultado;
54         }
55     }
56
57     public static void b(String s, int base) throws MiExcepcion{
58         int resultado;
59         try{
60             System.out.print("El_valor_numerico_de_"+s);
61             resultado= c(s,base);
62         }
63     }
64 }
```

Tu dosis diaria de Wall Street en solo 5 minutos.

[Suscríbete](#)

(y cómete a Wall Street)



Desarrollo de Software - Grupo A

7

```
62     System.out.print("en_base_10_es_"+resultado);
63 }
64 catch(MiOtraExcepcion e){ //Punto 2
65     System.out.println("MiOtraExcepcion:"+e.getMessage());
66     System.out.println("Manejado_en_el_punto_2");
67 }
68 finally{
69     System.out.print("\n");
70 }
71
72 }
73 }
74
75     public static void a(String s, int base){
76         try{
77             b(s,base);
78         }
79         catch(MiExcepcion e){//Punto 1
80             if(e instanceof MiSubExcepcion)
81                 System.out.print("MiSubExcepcion:");
82             else
83                 System.out.print("MiExcepcion:");
84             System.out.println(e.getMessage());
85             System.out.println("Manejado_en_el_punto_1");
86         }
87     }
88
89
90     public static void main(String[] args) throws MiExcepcion{
91         int base=10;
92         int max;
93         String s;
94         try{
95             s=args[0];
96             max= s.length();
97             if (max == 0) {
98                 throw new NumberFormatException("null");
99             }
100        }
101        catch(ArrayIndexOutOfBoundsException e){
102             System.out.println("Debes_especificar_un_argumento");
103             return;
104         }
105         a(s,base);
106     }
107 }
```

que debes saber

Los mercados la semana ha ido bastante «difi» rébete ha sido una semana de incertidumbre nos hacen pensar que la situación va a estar movida y inestable. La diferencia entre mercados ya que se recorren un volumen más acuero en estos frontes es histórico.

Stock de hoy**El ETF Favorito de Greta Thunberg**

Hoy no vamos a evaluar un stock en un ETF, que tiene mucho jugo en orden de cambios y sigue rozando los \$35K, evolucionando Energía Limpia que invita al Sí

examen-2014-junio-resultados.pdf



Anónimo



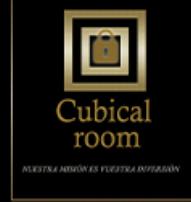
Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada


Cubical room
TU ESCAPE ROOM EN GRANADA



613001227
CALLE ARABIAL, Nº 132


cubicalroom.es
LA JUNGLA JUKANGI
¿ Podréis desafiar el reto de Jukangi..?
DESCUBRE LA SALA

Tu dosis diaria de Wall Street en solo 5 minutos.

Suscríbete



(y cómete a Wall Street)



1

Examen final

Nombre:

13-06-2014

I- Cada respuesta incorrecta restará 1/2 de la puntuación obtenida por cada respuesta correcta.

1. **(0,5)** Seleccionar la única respuesta correcta a la siguiente cuestión sobre el concepto de *patrón de diseño* en el desarrollo de software:
 - (a) Pueden ser un esquema algorítmico para resolver un problema que se repite.
 - (b) Las clases de un patrón de diseño no se pueden modificar (p.e.: necesito pasar un parámetro extra cuando invoco ciertos métodos de una clase del patrón) si lo hago, estaría proponiendo un patrón diferente.
 - (c) Un patrón propone una solución a un tipo de problema en un determinado contexto.
 - (d) Cada patrón es una estructura de clases fija que se adapta al desarrollo de una determinada aplicación o sistema-software.
2. **(0,5)** Seleccionar la única respuesta correcta a la siguiente cuestión sobre las diferencias entre patrones de diseño y marcos de trabajo (*frameworks*) en sistemas software:
 - (a) Los patrones de diseño poseen un menor grado de abstracción que los marcos de trabajo.
 - (b) Los patrones se pueden modificar para adaptarlos a las condiciones de problemas concretos, mientras que los marcos de trabajo se han de utilizar sin modificación.
 - (c) Los marcos de trabajo son menos especializados que los patrones de diseño.
 - (d) En general, los marcos de trabajo son elementos arquitectónicos más pequeños que los patrones.
3. **(0,5)** Indicar para cada uno de los siguientes ejemplos de sistemas a qué tipo de sistemas (S, P o E) pertenece:
 - (a) Un sistema de predicciones meteorológicas. **P**
 - (b) Un sistema de reserva de vuelos y hoteles. **E**
 - (c) Un programa para jugar al ajedrez. **P**
 - (d) Un programa que utiliza el polimorfismo para calcular la inversa de matrices para diferentes tipos de elementos: reales, complejos, enteros de diferentes precisiones. **S**
 - (e) Un sistema para control automático de vuelo de una aeronave. **P**
 - (f) Un sistema informático para modelar la Economía Mundial. **E**
4. **(0,5)** Seleccionar la única alternativa incorrecta a la siguiente cuestión sobre el proceso sistemático de prueba de una aplicación Web:
 - (a) Revisar el modelo de contenidos
 - (b) Comprobar el modelo de interfaz respecto de la inclusión de todos los casos de uso
 - (c) Detectar errores en el modelo de requisitos
 - (d) Ejercitarse la navegación del modelo de interfaz de usuario
 - (e) Detectar errores de navegación en el modelo de diseño
 - (f) Pruebas unitarias de componentes funcionales

WUOLAH

- (g) Revisar la facilidad de navegación a través de toda la arquitectura software de la aplicación
- (h) Pruebas de seguridad y robustez de la aplicación o de su entorno (p.e.: buscar enlaces rotos antes de instalar las páginas)
- (i) Comprobar compatibilidad con plataformas y sus configuraciones
- (j) Pruebas de rendimiento
5. (0,5) Seleccionar la única alternativa correcta a la siguiente cuestión sobre mantenimiento del software:
- (a) El mantenimiento *perfectivo* no necesariamente se realiza para detectar posibles fallos en el software
- (b) Los costes de desarrollo (implementación) de un sistema software serán siempre superiores al del mantenimiento del software
- (c) El que un software sea más o menos mantenible depende de factores internos y sólo se puede determinar con el uso prolongado del sistema
- (d) Según el estudio de Lienz y Swanson, el esfuerzo mayor de mantenimiento de un sistema software se emplea en realizar *mantenimiento correctivo*
6. (0,5) Seleccionar la única alternativa correcta a la siguiente cuestión sobre evolución del software:
- (a) El mantenimiento del software posee un ciclo de proceso independiente del ciclo de desarrollo del software
- (b) Los mantenimientos perfectivo y correctivo del software *deterioran* a un sistema software
- (c) El modelo predictivo de Belady–Lehman expresa la relación entre el coste de desarrollar el sistema (p), la complejidad del software (c) y la buena documentación (d) del mismo, que se resume con la siguiente ecuación $M = p + K \times c - d$. La constante empírica K se calcula después de la instalación y configuración para una plataforma software concreta. **Tener en cuenta que el valor de K depende fundamentalmente del entorno. Se determina comparando el esfuerzo de desarrollo que presenta nuestro modelo de sistema software con las relaciones de esfuerzo que se dan en varios proyectos de desarrollo reales similares. Por consiguiente, K no se calcula para una plataforma concreta sólo.**
- (d) El factor (SU) de valoración de la comprensión del código para el correcto mantenimiento del software según el modelo COCOMO depende fundamentalmente de la alta cohesión y bajo acoplamiento entre componentes (módulos, clases, paquetes) del sistema
7. (0,5) Seleccionar la única respuesta correcta a la siguiente cuestión sobre factores internos que dificultan el mantenimiento de un sistema software:
- (a) El paradigma de programación utilizado (PDO, imperativo, concurrente, etc.) no afecta a la dificultad de mantenimiento del software
- (b) Lo único importante para que un sistema software se pueda mantener sin grandes dificultades es que el número de McCabe o *ciclomático* sea lo más bajo posible
- (c) La herencia y el polimorfismo podrían ayudar a deteriorar el sistema tras la realización de acciones de mantenimiento (correctivo, adaptativo, etc.)
- (d) El número ciclomático se puede determinar también mediante inspección del código, ya que es igual al número de bucles + 2

**ENGINEERING
THE
FUTURE**

¿Crees en el poder de la tecnología para cambiar el mundo?

¡Indra te espera en Twitch!

No faltes a la experiencia digital en la que conocerás las tecnologías, los proyectos y las personas que están transformando el futuro.

¡La defensa, la movilidad y el sector aeroespacial necesitan tu talento!
Asómate al futuro y haz ingeniería del mañana, hoy.



7 de junio - 17 hs
¡Inscríbete!



indra

II- Ejercicios sobre la teoría impartida

1. (1,0) Rehacer el diagrama de clases de la figura 1 para que se adapte al patrón explicado *Jerarquía General*

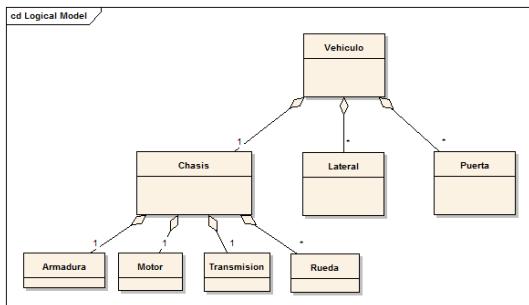


Figura 1: Jerarquía de partes de un coche

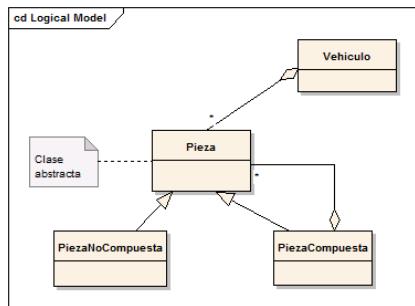


Figura 1: Solución utilizando el patrón Jerarquía General

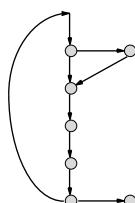
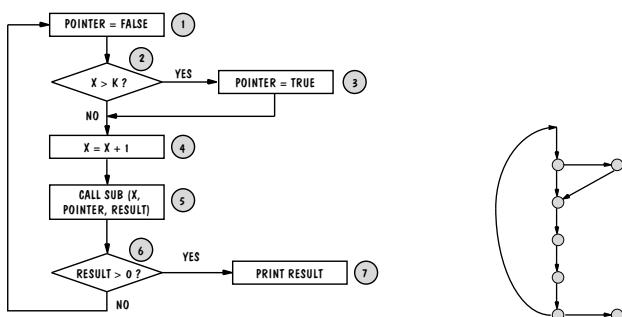


Figura 2: (a) Diagrama de flujo de un programa simple (b) Grafo orientado equivalente

2. (1,0) Considerar un diagrama de flujo de un programa como un grafo dirigido en el cual los rombos y cajas del programa se consideran como nodos y las flechas de flujo lógico entre ellos como aristas orientadas del grafo. Por ejemplo el programa de la figura 2 siguiente puede graficarse como se muestra al lado. Calcular el número de McCabe o ciclomático que se infiere para el programa representado por el diagrama de flujo.

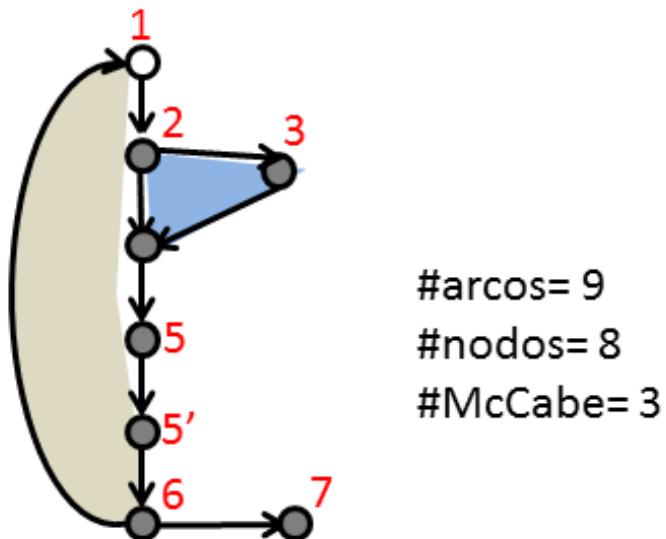


Figura 3: Solución

Por último, probar que la prueba de caminos es equivalente a encontrar todos los caminos posibles a través del grafo. Para demostrar la fórmula de McCabe: $M = E - N + 2$ se puede utilizar un grafo en el cual cada nodo de salida vuelve con un arco hacia el nodo de entrada del componente. En tal caso el grafo será ahora fuertemente conexo y la complejidad ciclomática del programa es igual al número ciclomático del grafo (conocido como el primer número de Betti), que viene dado por la ecuación: (1) $M = E - N + 2 * P$, donde P es su número de componentes conexos. Esto se podría ver como el cálculo del número de ciclos linealmente independientes que existen en el grafo (aquellos ciclos que no contienen a otros ciclos dentro) fuertemente conexo, ya que tiene que existir al menos 1 ciclo por cada punto de salida porque existe el arco hacia atrás hasta el punto de entrada del componente. Evidentemente si eliminamos el arco desde el nodo de salida al de entrada de cada componente, los ciclos se convierten en caminos linealmente independientes dentro del grafo total.

Concluyendo que, para un programa simple (método o subrutina), el parámetro P siempre es igual a 1 y se obtiene la fórmula de McCabe:(2) $M = E - N + 2$ a partir de (1).

3. **(1,0)**Una clase *singleton* que se denomina MotorJuego ha de provocar la creación de animales (objetos) que pertenezcan a distintos continentes en un juego de ordenador. La idea para implementar el juego consiste en que, dependiendo del continente al que pertenezca el país seleccionado por el usuario, se crean los animales apropiados (por ejemplo: leones y gacelas en Tanzania; pumas y castores en Canada). Utilizando el patrón de diseño *factoría* dibujar un diagrama de clases en el que han de aparecer, al menos, las siguientes clases: MotorJuego, Animales, AnimalesAfrica, AnimalesAmerica, CreadorAnimales (incluye el método factoría crearAnimales (Pais XXX)), CreadorAnimalesPaisXXX (que implementa el método factoría).

Tu dosis diaria de Wall Street en solo 5 minutos.

Suscríbete



(y cómete a Wall Street)



Desarrollo de Software - Grupo A

5

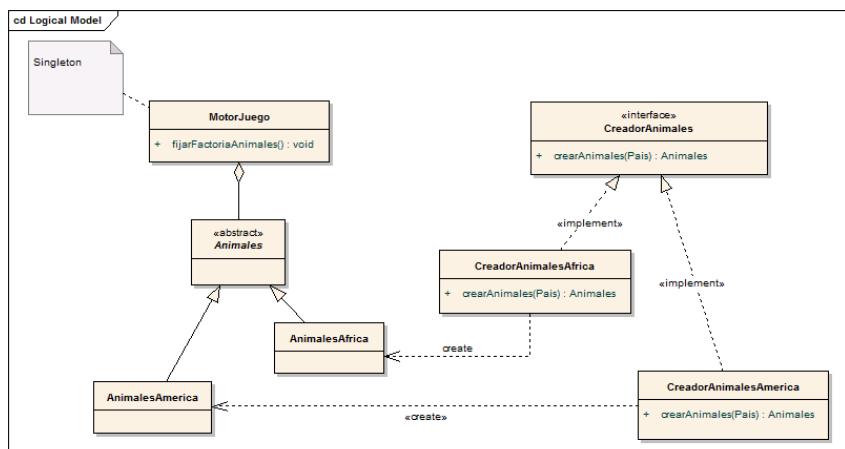


Figura 4: Solución

III- Preguntas cortas

- (0,5)** Describir el papel de las *metodologías de diseño OO, marcos de trabajo y patrones de diseño* en el análisis y diseño de un sistema software. Las metodologías de diseño orientadas a objetos (OO) facilitan el análisis y diseño de un sistema software y mejoran su calidad final gracias a los principios de alta cohesión y bajo acoplamiento que propician las citadas metodologías (Grady Booch, 1987) en el desarrollo de componentes software.

Los patrones de diseño refuerzan las metodologías y técnicas OO ya que proporcionan posibles soluciones a problemas comunes de diseño de software, lo que permite recurrir a dichas soluciones “prefabricadas” para ahorrar tiempo en la búsqueda de una solución a un problema específico.

Los marcos de trabajo (*frameworks*), por su parte, nos permiten definir un esqueleto (conjunto de clases) que podemos adaptar para obtener el software concreto que necesitamos construir.

- (1,0)** Indicar el patrón de diseño más apropiado para ser aplicado a la solución de los siguientes problemas y justificar dicha elección brevemente:

(a) Estamos desarrollando una marco de trabajo para usarlo en la presentación de una interfaz de usuario con datos de cotizaciones de bolsa. Queremos obtener las cotizaciones tan pronto estén disponibles. También que nuevas cotizaciones activen determinados cálculos financieros y ambas cosas a la vez, además de hacer que las cotizaciones sean transmitidas sin cables a una red de *pgers*. ¿Cómo diseñar el marco de trabajo de tal forma que varias partes diferentes del código de la aplicación pudieran reaccionar de manera independiente a la llegada de nuevas cotizaciones?

Patrón de diseño: **Observador**

Justificación: *CotizarAcciones()* sería el observable y las otras clases (cálculos financieros, etc.) serán sus observadores.

- Queremos incluir nuevas operaciones de objetos *PolygonRegular*, que distorsionen los objetos-polígonos, de tal forma que dejen de ser regulares. ¿Cómo hacemos para permitir la ejecución de dichas operaciones sin que se levanten excepciones en el programa?

WUOLAH

Patrón de diseño: **Inmutable**

Justificación: **Las operaciones que distorsionan un polígono regular devuelven instancias de otra clase, que no serían polígonos regulares.**

- (c) Se está construyendo un catálogo de productos que vende la empresa en la que trabajamos. Queremos reutilizar algunas clases proporcionadas por los suministradores, que no podemos modificar. ¿Cómo podíamos asegurarnos que las clases suministradoras puedan seguir utilizándose y mantener el polimorfismo de los objetos de nuestra aplicación?

Patrón de diseño: **Adaptador**

Justificación: **Se puede crear una clase de tipo “adaptador”, tal que sus métodos deleguen en los de las clases de los suministradores.**

- (d) Nuestro programa manipula imágenes muy pesadas. ¿Cómo podemos diseñar un programa que sólo cargue las imágenes en memoria cuando las necesite?

Patrón de diseño: **Proxy**

Justificación: **La mayoría del tiempo se mantendrá una instancia del proxy–imagen en memoria. Proxy–imagen proporciona una interfaz idéntica a la de un objeto imagen de la aplicación, pero difiere del objeto real sólo si se ejecutan operaciones que no modifiquen el estado del objeto.**

- (e) Hemos creado un subsistema con 25 clases. Sabemos que los componentes del resto del sistema accederán, en promedio, a sólo 5 métodos de nuestras clases. ¿Cómo podemos simplificar la vista que posee el resto del sistema de nuestro subsistema?

Patrón de diseño: **Façade**

Justificación: **Se programa una clase de este tipo con una interfaz pública de sólo 5 métodos.**

3. (1,0) (a) Explicar brevemente la ecuación fundamental del modelo predictivo de costes de mantenimiento de Belady-Lehman $M = P + K \times c - d$ y contestar a la siguiente cuestión justificándola. color blue La ecuación anterior indica que la predicción de costes debe tener en cuenta 3 factores fundamentales: una estimación sobre el esfuerzo necesario para el desarrollo (análisis, evaluación, diseño, codificación y pruebas) del sistema (P); la complejidad ciclomática (c) del propio producto software, multiplicado por una constante empírica que depende del entorno del sistema (K); la familiaridad que poseen los técnicos de mantenimiento con el software que han de mantener (d). (b) cuál de las siguientes decisiones eliges como más costo–efectiva para el mantenimiento futuro de un sistema software, que acaba de ser entregado, que posee una documentación poco clara y alta complejidad ($c = 48$, ver Tabla 1) Elegir la (a) :

(a) Pedir una nueva documentación (10,000 EUR) aumentará la familiaridad con el software que se ha de mantener y, por tanto, podría reducir considerablemente los costes derivados de su mantenimiento. No es la inversión más cara, considerando los beneficios que reportaría.

(b) Devolver el software para que lo refactoricen y bajen la complejidad ciclomática un 50% (hora programador= 80 EUR, 10,000 líneas de código, 40 paquetes y 120 componentes) Puede resultar muy cara de llevar a cabo: 80,000 EUR para revisar un código de 10,000 líneas con el número ciclomático= 48. Los datos que nos están dando demuestran que se trata de un software muy complejo (riesgo “alto” según la tabla) y extenso. Se requieren

muchas horas de trabajo de programadores para reducir su complejidad en un 50%. Es posible que también haya que modificar el diseño del software con lo que los costes se dispararían. Además, incluso después de esta modificación, la documentación seguiría siendo deficiente, por lo que persistirían las dificultades para conseguir un buen mantenimiento del sistema.

- (c) Cambiar el sistema operativo, la constante K disminuye su valor el 20%. Es menos efectiva que la opción (b) anterior, ya que la complejidad se reduce sólo en un 20% en lugar del 50% si se adopta la solución anterior. Esta solución no tendría ningún impacto en el esfuerzo (P) de desarrollo ni en la familiaridad (d) con el sistema, por consiguiente los costes de mantenimiento se verían poco rebajados.

complejidad (c)	riesgo	tiempo prueba (horas)
1–10	bajo	10/1000 líneas
11–20	moderado	50/1000 líneas
21–50	alto	100/1000 líneas
51+	imposible tests	—

Tabla 1: Valoración de riesgos según el número ciclomático

IV- Supuesto Práctico

1. (1,0) Considerando el siguiente programa Java cuando se ejecuta: (a)sin argumentos, (b) con un objeto *String* como argumento y (c) con argumentos enteros: 0, 1, 2 y 99. Explicar cómo funciona la gestión de las excepciones que se han implementado (indicar el orden de ejecución de los bloques) en dicho programa y la salida que producirá en cada caso. [Solución en la relación de ejercicios resueltos del Tema 4: ejercicio #63.](#)

```

1 class MiExcepcion extends Exception{
2     public MiExcepcion(){ super(); }
3     public MiExcepcion(String s){ super(s); }
4 }
5 class MiOtraExcepcion extends Exception{
6     public MiOtraExcepcion(){ super(); }
7     public MiOtraExcepcion(String s){ super(s); }
8 }
9 class MiSubExcepcion extends MiExcepcion{
10    public MiSubExcepcion(){ super(); }
11    public MiSubExcepcion(String s){ super(s); }
12 }
13 public class throwtest{
14     public static void main(String argv[]){
15         int i;
16         try{
17             i= Integer.parseInt(argv[0]);
18         }
19         catch(ArrayIndexOutOfBoundsException e){
20             System.out.println("Debes_especificar_un_argumento");
21             return;
22         }
23         catch(NumberFormatException e){
24             System.out.println("Debes_especificar_un_argumento_entero");
25             return;
26         }
27         a(i);
28     }
29     public static void a(int i){
30         try{
31             b(i);
32         }
33         catch(MiExcepcion e){ //Punto 1
34             if(e instanceof MiSubExcepcion)
35                 System.out.print("MiSubEXcepion:");
36             else
37                 System.out.print("MiExcepcion:");
38             System.out.println(e.getMessage());
39             System.out.println("Manejado_en_el_punto_1");
40         }
41     }
42     public static void b(int i) throws MiExcepcion{
43         int result;
44         try{
45             System.out.print("i="+i);
46             resultado= c(i);
47             System.out.print("c(i)="+resultado);
48         }
49         catch(MiOtraExcepcion e){ //Punto 2
50             System.out.println("MiOtraExcepcion:"+e.getMessage());
51             System.out.println("Manejado_en_el_punto_2");
52         }
53         finally{
54             System.out.print("\n");
55         }
56     public static int c(int i) throws MiExcepcion, MiOtraExcepcion{
57         switch(i){
58             case 0: throw new MiExcepcion("entrada_demasiado_baja");
59             case 1: throw new MiSubExcepcion("entrada_todavia_muy_baja");
60             case 99: throw new MiOtraExcepcion("entrada_muy_alta");
61             default: return i*i;
62         }
63     }
64 }
```



Tu dosis diaria de Wall Street en solo 5 minutos.

Suscríbete



(y cómete a Wall Street)



Desarrollo de Software - Grupo A

9

63
64



que debes saber

Los mercados la semana ha ido bastante «dilatado» ha sido una semana de incertidumbre. Los expertos nos hacen pensar que la situación económica mundial seguirá siendo incierta y no se sabe si la situación política a nivel mundial ya que los acuerdos entre países están siendo retrasados.

Stock de hoy

El ETF Favorito de Greta Thunberg



Hoy no vamos a evaluar un stock en un ETF, que tiene mucho jugo en orden de cambios y sigue rotando los \$35K, evolucionando Energía Limpia que invita al S&P 500.

WUOLAH

Parcial2DS.pdf



AdriMartin



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

Correcto.

¡Pruébalo aquí!

¿ES CON V O CON B?

Con C, de Correcto.



#escribeunfuturomejor

Parcial 2 DS

Tipo Test

1. Importancia AS (marcar la incorrecta):

- Las decisiones de diseño que se toman en la elaboración de las AS son muy importantes para conseguir que el sistema final satisfaga sus requisitos no funcionales críticos.
- Sirven para obtener un “diseño de clases” de la aplicación o el sistema software objetivo.
- La elaboración de una AS facilita que el sistema final posea determinados atributos de calidad.
- Constituye una parte importante de la documentación del diseño
- Contar con una AS es fundamental para poder llevar a cabo con éxito el conjunto de tareas denominadas prototipado rápido.

2. Estructura de AS (marcar la correcta):

- Las estructuras modulares (capas layers) facilitan el atributo de calidad denominado Modificabilidad del sistema, es decir, el atributo que tiene que ver con la facilidad para realizar cambios.
- Las estructuras de asignación (Despliegue Deployment) facilitan el atributo de calidad llamado Portabilidad, es decir, la utilización del mismo software en diferentes entornos de ejecución.
- La estructura Componentes y Conectores (C&C) denominada servicio facilita el atributo llamado Interoperabilidad, es decir, propician trabajar con otros productos o sistemas, presentes o futuros, sin conocer su implementación o tener que asumir restricciones de acceso.
- La estructura modular denominada relación de uso facilita el atributo de calidad llamado Delegación, es decir, evaluar un campo de un objeto (el receptor) en el contexto de otro objeto original (remitente).
- La estructura Componentes y Conectores (C&C) denominada concurrencia facilita el atributo de calidad llamado Interoperabilidad.

3. Marcar la incorrecta:

- Seguridad: los componentes que lo cumplen son capaces de recuperarse de errores y fallos del sistema.
- Disponibilidad: tiene que ver con el relevo de componentes software en presencia de un fallo del sistema.
- Usabilidad: aislar los detalles de la interfaz de un componente de su implementación interna y del resto del sistema.
- Interoperabilidad: controlar a los elementos del sistema que son responsables de las comunicaciones externas.
- Delegación: los componentes que lo cumplen son capaces de pasar “algo” a otras entidades o, en un contexto de objetos, evaluar un miembro del objeto que recibe en el contexto del objeto que origina que pase.

4. Estrategias de la composición de un ServicioWeb (marcar la correcta):

- La orquestación de WS promueve un método de diseño top-down de aplicaciones de software orientado a servicios.
- Un servicio de orquestación de WS siempre posee el atributo de ser Colaborativo, es decir, un software de aplicación diseñado para ayudar a lograr sus objetivos a personas de distintas organizaciones involucradas en una tarea común.
- La composición de servicios solo consiste en reunir virtualmente a varios SW, de tal forma que las aplicaciones cliente perciban dicha composición como un único servicio.

-Respecto a las actividades que participan en los flujos de trabajos de las orquestaciones de SW, una actividad puede completarse cuando se cumpla al menos uno de los requisitos de sus enlaces salientes.

-Antes de comenzar cualquier actividad han de cumplirse los requisitos de sus enlaces entrantes.

5. Patrones arquitectonicos y predicción de la calidad software (marcar la incorrecta):

-Podemos anticipar la calidad del software considerando únicamente el nivel de satisfacción de los requisitos funcionales abordados por los patrones arquitectónicos específicos.

-Podemos anticipar la calidad de un software si analizamos el grado de posible cumplimiento de las características de calidad ISO.

-Los patrones arquitectónicos tienen una importante relación con la calidad del software de la aplicación final porque éste depende principalmente de las características de los patrones elegidos.

-La utilización de los patrones arquitectónicos tiene una importante relación con la facilidad para el mantenimiento y evolución del software.

-Los servicios que exponen su estado interno pueden ser descubiertos más fácilmente por otros servicios de las aplicaciones que se desarrollan según la filosofía de SOA.

6. ISO y patron Interceptor (marcar la correcta):

-La utilización de este patrón siempre producirá aplicaciones software excelentes.

-El rendimiento de las aplicaciones cliente podría degradarse si se dan las condiciones dinámicas en el programa para encadenamiento de interceptores excesivamente largos.

-Los interceptores nunca pueden ser reutilizados en una aplicación diferente de la que se programaron.

-Utilizar un interceptor en el desarrollo de una aplicación afecta necesariamente a las clases del marco de trabajo donde se vayan a ejecutar, es decir, convierte la aplicación cliente en no-interoperable.

-Los interceptores han de ejecutarse siempre secuencialmente en el software de las aplicaciones que los utilicen, es decir, no puede darse la ejecución concurrente de varios interceptores en una misma aplicación software.

Ejercicios

1. Ventajas e inconvenientes de utilizar mecanismo de reflexión en JAVA o hacer uso del polimorfismo de los lenguajes de POO para programar un método para manejar cadenas que las aplicaciones cliente envían al método `handleMessageFromClient(String nombre_método)`:
Para la clase:

```
public final class <T> extends Object implements Serializable,  
GenericDeclaration, Type, AmonteElem{...}
```

2. Para el patrón Reflection: explicar qué significa cada nivel de objetos del diagrama. Programar un pequeño ejemplo.

Preguntas Cortas

1. Por qué la utilización de AS antes de iniciar un proyecto da los siguientes beneficios:
propiciar/inhibir atributos de calidad; predicción temprana de los atributos de calidad; facilitar el razonamiento sobre el sistema y sus cambios; documentación que propicie la comunicación entre las partes; soporte idóneo para decisiones de diseño tempranas; conjunto de restricciones de implementación del software; base fundamental para realizar prototipado rápido.

2.Indicar 5 principales recomendaciones de AS y explicar brevemente sus significado.

Ejercicio Largo

Se pretende diseñar la arquitectura de software de una aplicación interactiva como un Servicio Web desplegable en un proveedor de PaaS y también una versión app de la misma aplicación para dispositivos móviles con Android. Una aplicación Web que ofrezca información geográfica (Google Maps), cultural, etc., relativa a los municipios y su relación con la historia del agua en la provincia de Granada.

En una primera pantalla (ver Figura 2) tendríamos una imagen de mapa cartográfico en el que se localizan los municipios e hitos de interés (museos, rutas de senderismo, embalses) de la provincia de Granada. En esta misma pantalla tendríamos un botón de acceso que mostrase el mapa de municipios etiquetado (ver Figura 3), utilizando para ello Google Maps etiquetados convenientemente.

(ejercicio23 de la relacion)

Se pide: DCU Servicio Web (cliente), DCU del cliente app, Diagrama de clases del Servicio Web(cliente) no hay que hacer el REST, Diagrama de clases de la app; diagrama de secuencia por parte de un usuario en PC y móvil

parcial3.pdf



AdriMartin



Desarrollo de Software (Especialidad Ingeniería del Software)



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

Correcto.



**NO ERES TÚ, ES TU
ORTOGRAFÍA**

#escribeunfuturomejor

¡Quiero probarlo ya!





Parcial 3 DS

Tests

1.MDA (marcar la incorrecta):

- MDA se sustenta en las tecnologías UML y MOF para conseguir auténticos modelos ejecutables de los sistemas.
- Existen varios enfoques para abordar un desarrollo dirigido por modelos, uno de ellos es MDA (no es el único).
- Elimina totalmente la necesidad de programar algoritmos en el desarrollo de sistemas y aplicaciones.
- UML no especifica la sintaxis textual concreta de ningún lenguaje para implementar acciones y actividades para representar el comportamiento del sistema o las interacciones entre sus componentes.
- Cada herramienta para realizar MDA/MDD han de proporcionar un lenguaje de acciones específica.

2.Marcar la correcta:

- Un PIM de un determinado sistema no puede contener diagramas de clases, ni diagramas de estado, ni diagramas de actividad o algún otro diagrama específico del dominio de aplicación.
- Un PIM podría dar lugar a varios PSM.
- Un PIM de un sistema basado en Web podría describir páginas html y los programas para interpretar datos de los formularios.
- El CIM genera automáticamente el código del sistema.
- Solo una vez se puede transformar 1PIM a 1PSM para cada plataforma específica.

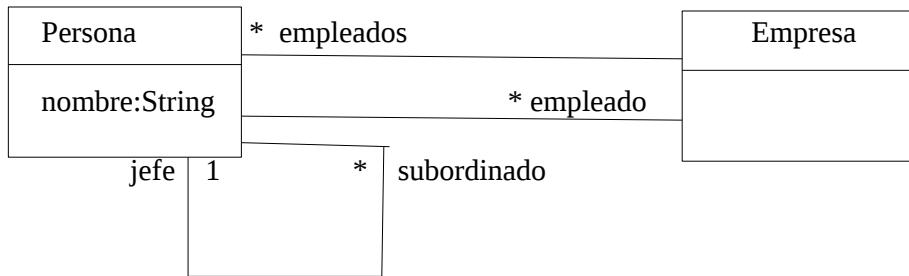
3.Marcar la incorrecta:

- Las herramientas de transformación entre el PIM y el PSM se sustituye por un conjunto de reglas que se combinen entre ellas para conseguir cambiar un modelo de lenguaje fuente en otro lenguaje distinto.
- La denominada definición de transformación en MDA se trata de una definición acerca de como se debe transformar cualquier modelo descrito en lenguaje origen o inicial a otro modelo en el lenguaje destino o final.
- Ha de existir un metalenguaje común a los lenguajes origen, destino y definición de transformación.
- La definición de una transformación es un conjunto de reglas escritas en un lenguaje de transformaciones entre modelos.

4.Marcar la correcta sobre lenguajes origen y destino:

- Existen limitaciones en cuanto a los lenguajes origen y destino para los que aplicar las transformaciones MDA.
- Los lenguajes origen y destino no pueden coincidir.
- El que el origen y el destino sean un mismo lenguaje no garantiza que las construcciones de los modelos inicial y final tengan el mismo significado.
- A efectos de aplicar transformaciones MDA, los perfiles de UML son todos el mismo lenguaje.
- Se puede utilizar restricciones de cualquier tipo (OCL, Statecharts, ..) para especificar un comportamiento funcional.

5.Marcar la correcta sobre el concepto alcance y la figura:



-El rol empleados define el conjunto de todos los objetos de la clase Persona que trabajan en cualquier empresa.

-El rol empleado se puede interpretar como una restriccion referida a que una persona no puede ser empleada de mas de 1 empresa.

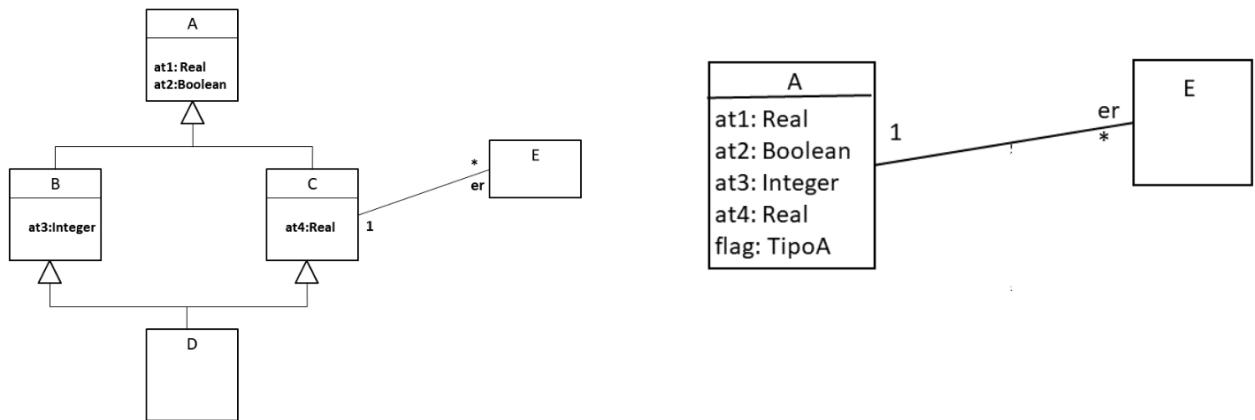
-La asociacion cuyo rol es subordinacion esta mal definida porque se necesitan que existan subordinantes distintos antes de la designacion de un jefe.

-Los atributos son propiedades de las clase del diagrama, pero los denominados roles no lo son.

-2 personas con el mismo valor del atributo nombre podrian ser empleados de la misma persona.

-1 persona con el rol jefe podria tener tambien el rol subordinacion.

6. Marcar la correcta sobre la fusion:



-La transformacion del diagrama es correcta sin ninguna restriccion adicional sobre los roles que se les puede asignar a los objetos de A.

-A todo objeto de la clase A le corresponde 0 o mas objetos de la clase E, que asumen el rol 'er' (el diagrama se puede considerar correcto).

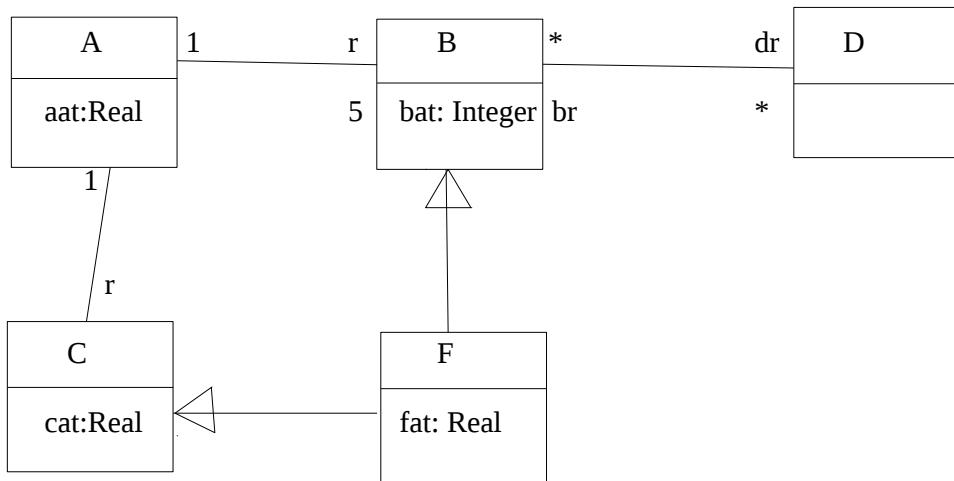
-A varios objetos de la clase A se les podria asignar el mismo rol 'er' (diagrama completo).

-Solo los objetos de clase A con un valor del atributo 'flag' igual a 'isC' se les podria asignar roles (diagrama derecho incompleto).

-Solo los objetos de la clase A con un valor atributo 'flag' distinto de isA o isB se les podria asignar roles (diagrama derecho incompleto).

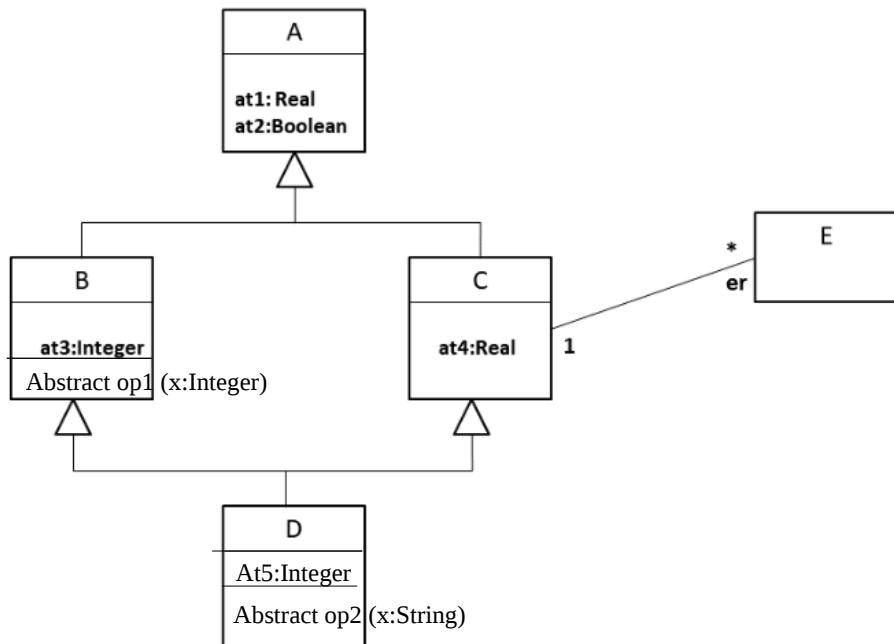
Ejercicios sobre la teoria

1. Aplicar transformaciones entre modelos que sean necesarias para mejorar la estructura de clase de la figura. Ademas especifica formalmente los alcances de todas las propiedades de ambos modelos y relacionar cada alcance del modelo inicial con el modelo final.



2. El PIM cumple la implementacion del PSM:

- Explicar los fallos que hay
- Dibujar el diagrama corregido



Preguntas Cortas

- Explicar el significado de los criterios de agrupacion estudiados necesarios para elaborar una arquitectura de diseño con modulos altamente cohesivos y bajo acoplamiento entre estos de un sistema informatico.
- Explicar el concepto de Factoria de software