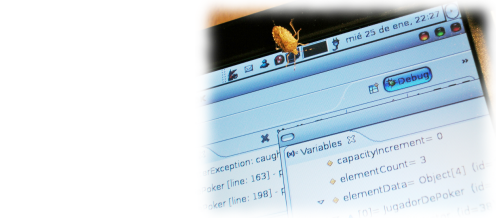
Tema 6: Introducción al Diseño

BLOQUE II:
ANÁLISIS Y DISEÑO DE LOS SISTEMAS SOFTWARE

Segundo curso de Grado en Ingeniería Informática
Curso 2014-2015

Javier Sánchez Monedero
jsanchezm@uco.es
http://www.uco.es/users/i02samoj



Índice

Índice

1.

2. Introducción

Introducción

Advertencia

Advertencia
Estos apuntes no contienen todo el material necesario respecto al tema. Intentan dar una visión de conjunto y proporcionar los conocimientos básicos para que podáis consultar la bibliografía y manuales de referencia, así como material complementario colgado en Moodle.

Objetivos

- Conocer los principios de diseño de arquitectura software.
- Conocer la existencia de patrones de diseño y explorar algunos relevantes, así como su aplicación.

El diseño software

- En general, el **diseño** (software o no) se refiere al proceso de implementar soluciones software para uno o más problemas.
- Dentro del desarrollo software, normalmente se usa el término ***diseño software*** para referirse al ***diseño de la arquitectura software***.
- En metodologías de ingeniería del software podréis observar que diseño puede ser **una actividad** del proceso, o que va conjuntamente con la actividad de análisis (**Análisis y Diseño** por ejemplo en OpenUP).

Parece razonable que la frontera entre analizar un problema (tras la identificación de requisitos) y diseñar solución no esté clara.

Buscando un consenso en la definición

Diseño de la arquitectura
Según [Pressman, 2010][Capítulo 9]:

> Se ha descrito al diseño como un proceso de etapas múltiples en el que, a partir de los requerimientos[1] de información, se sintetizan las representaciones de los datos y la estructura del programa, las características de la interfaz y los detalles del procedimiento.

[1] El libro de Pressman en castellano contiene innumerables barbaridades en la traducción

Table 1. Frequency of Common Concepts in Analyzed Definitions

Concept	Frequency
Design as a *process*	11
Design as creation	11
Design as *planning*	7
Design as a *physical* activity (or as including implementation)	7
System (as the object of the design)	7
Design as being *deliberate*, or having a *purpose, goal* or *objective*	7
Design as an *activity*, or a collection of activities	7
Design as occurring in an environment (or domain/situation/context)	7
Artifact, as the object of the design	5
Needs or *requirements*	5
Design as a *human* phenomenon	5
Design as *organizing*	4
Parts, components or elements	4
Constraints or *limitations*	3
Process (as the object of design)	2
Design as *creative*	2
Optimizing	2
Design as a *mental* activity	2
Resources	2

Figura 1: Términos más frecuentes al hablar de 'diseño', fuente [Ralph and Wand, 2009]

Según Freeman et al. [2004][2]:

> El diseño es una actividad que tiene que ver con la toma de decisiones importantes, con frecuencia de naturaleza estructural. Comparte con la programación el objetivo de abstraer una representación de la información y de las secuencias de procesamiento, pero en los extremos el grado de detalle es muy distinto. El diseño elabora representaciones coherentes y bien planeadas de programas, que se concentran en las relaciones de las partes en el nivel más alto y en las operaciones lógicas involucradas en los niveles bajos.

Los métodos de diseño del software provienen de los tres dominios del modelo de análisis. Los dominios de **datos**, **funciones** y **comportamiento** sirven como guía para la creación del diseño del software, más formalmente [Pressman, 2010]:

Arquitectura software
La arquitectura del software de un programa es la estructura o estructuras del sistema, lo que comprende a los componentes del software, sus propiedades externas visibles y las relaciones entre ellos.

Según Pressman [2010], la arquitectura no es el software operativo[3], es una representación que permite:

[2]Los autores tienen publicado un listado de patrones de diseño para Java 8 en https://github.com/bethrobson/Head-First-Design-Patterns

[3]Se debe tomar esta afirmación de R. Pressman de forma relativa, ya que en temas anteriores hemos mencionado que muchas metodologías hacen hincapié en el concepto de "arquitectura ejecutable" o similares.

1. Analizar la efectividad del diseño para cumplir los **requisitos** establecidos.
2. Considerar **alternativas arquitectónicas** en una etapa en la que hacer cambios al diseño todavía es relativamente fácil.
3. **Reducir los riesgos** asociados con la construcción del software.

Resumen de la actividad de diseño

Según Pressman [2010]:

- **¿Qué es?**. El diseño arquitectónico representa la **estructura** de los **datos** y de los **componentes** del programa que se requieren para construir un sistema basado en computadora. Considera el **estilo de arquitectura** que adoptará el sistema, la estructura y las propiedades de los componentes que lo constituyen y las interrelaciones que ocurren entre sus componentes arquitectónicos.
- **¿Quién lo hace?**. Frecuentemente el **ingeniero/a de software**. En grandes sistemas, el diseñador/a de la base de datos creará, por ejemplo, la arquitectura de datos.
- **¿Por qué es importante?** Se suele equiparar el diseño software a los planos de una casa: sin ellos resulta difícil comenzar la construcción. El diseño de la visión general o panorama del sistema.
- **Pasos**. El diseño de la arquitectura comienza con el diseño de los datos y continúa con la obtención de una o más representaciones de la estructura arquitectónica del sistema. Se analizan alternativas de estilos o **patrones arquitectónicos** para llegar a la estructura más adecuada para los requisitos del usuario y criterios de calidad.
- **Artefacto final.** El modelo de arquitectura incluye la arquitectura de datos y la estructura del programa, además de las propiedades y relaciones entre los componentes.
- **Revisión.** El resultado del diseño software debe debe ser consistente con los requisitos y entre si.

3. El diseño software

El diseño software

Objetivos del diseño

Objetivos del diseño [Irene T., 2013]:

- Implementar todos los requisitos explícitos.

- Ser la guía para quienes construyan, prueben y mantengan el código.
- Proporcionar una idea completa del software.

Actividades del diseño

Actividades [Irene T., 2013]::

- **Diseño de Datos**: diseño de archivos / tablas.
- **Diseño Arquitectónico**: define una estructura modular.
- **Diseño de Interfaz**: define cómo el software se comunica. consigo mismo, con los sistemas que operan con él y con los operadores que lo emplean.
- **Diseño procedimental o de las funciones**: diseño de los algoritmos.

Géneros del diseño

Pressman [2010] llama ***géneros*** del diseño a distintos **estilos de diseño** que dependen del ámbito o dominio del problema. Se enumeran los siguientes:

- **Inteligencia artificial**: Sistemas que simulan o incrementan la cognición humana, su locomoción u otros procesos orgánicos.
- **Comerciales y no lucrativos**: Sistemas que son fundamentales para la operación de una empresa de negocios.
- **Comunicaciones**: Sistemas que proveen la infraestructura para transferir y manejar datos, para conectar usuarios de éstos o para presentar datos en la frontera de una infraestructura.
- **Contenido de autor**: Sistemas que se emplean para crear o manipular artefactos de texto o multimedios.
- **Dispositivos**: Sistemas que interactúan con el mundo físico a fin de brindar algún servicio puntual a un individuo.
- **Entretenimiento y deportes**: Sistemas que administran eventos públicos o que proveen una experiencia grupal de entretenimiento.
- **Financieros**: Sistemas que proporcionan la infraestructura para transferir y manejar dinero y otros títulos.
- **Juegos**: Sistemas que dan una experiencia de entretenimiento a individuos o grupos.
- **Gobierno**: Sistemas que dan apoyo a la conducción y operaciones de una institución política local, estatal, federal, global o de otro tipo.
- **Industrial**: Sistemas que simulan o controlan procesos físicos.

- **Legal**: Sistemas que dan apoyo a la industria jurídica.
- **Médicos**: Sistemas que diagnostican, curan o contribuyen a la investigación médica.
- **Militares**: Sistemas de consulta, comunicaciones, comando, control e inteligencia (o C4I), así como de armas ofensivas y defensivas.
- **Sistemas operativos**: Sistemas que están inmediatamente instalados en el hardware para dar servicios de software básico.
- **Plataformas**: Sistemas que se encuentran en los sistemas operativos para brindar servicios avanzados.
- **Científicos**: Sistemas que se emplean para hacer investigación científica y aplicada.
- **Herramientas**: Sistemas que se utilizan para desarrollar otros sistemas.
- **Transporte**: Sistemas que controlan vehículos acuáticos, terrestres, aéreos o espaciales.
- **Utilidades**: Sistemas que interactúan con otro software para brindar algún servicio específico.

Estilos arquitectónicos según R. Pressman

Una taxonomía general de Pressman nos permite definir los siguientes **estilos** (no excluyentes entre si)[4]:

- **Arquitecturas centradas en los datos**. En el centro de esta arquitectura se halla un almacenamiento de datos (como un archivo o base de datos) al que acceden con frecuencia otros componentes que actualizan, agregan, eliminan o modifican los datos de cierto modo dentro del almacenamiento. Hoy en día existen tecnologías para el desarrollo de sistemas distribuidos centrados en los datos, como *Data Distribution Service* [Object Management Group, 2014].
- **Arquitecturas de flujo de datos**. Esta arquitectura se aplica cuando datos de entrada van a transformarse en datos de salida a través de una serie de componentes computacionales o manipuladores. El flujo puede tener una o varias líneas de transformaciones.
- **Arquitecturas de llamar y regresar.** Fácil de modificar y escalar, presenta varios subestilos:
  - *Arquitecturas de programa principal/subprograma*. Esta estructura clásica de programa descompone una función en una jerarquía de control en la que un programa "principal" invoca cierto número de componentes de programa que a su vez invocan a otros

[4] Recuerda las explicaciones y ejemplos en pizarra.

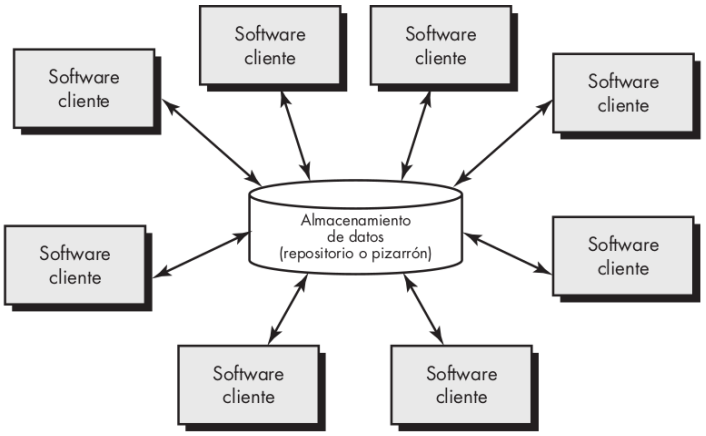


Figura 2: Arquitectura centrada en datos, fuente [Pressman, 2010].

- *Arquitecturas de llamada de procedimiento remoto*. Los componentes de una arquitectura de programa principal/subprograma están distribuidos a través de computadoras múltiples en una red.

- **Arquitecturas orientadas a objetos**. Los componentes de un sistema incluyen datos y las operaciones que deben aplicarse para manipularlos. La comunicación y coordinación entre los componentes se consigue mediante la transmisión de mensajes.

- **Arquitecturas en capas.** Se definen varias capas diferentes, y cada una ejecutan operaciones desde más alto nivel a más bajo nivel (lenguaje máquina o señales físicas en comunicaciones).

Más estilos arquitectónicos

Existen más estilos que puedes consultar en Wikipedia[5].

Microsoft también tiene un buen resumen de estilos arquitectónicos dentro de una web de documentación titulada *Microsoft Application Architecture Guide* [**?**].

Principios del diseño

Según los apuntes de Irene T. [2013], algunos principios generales de diseño son:

[5] http://en.wikipedia.org/wiki/Software_architecture_styles_and_patterns

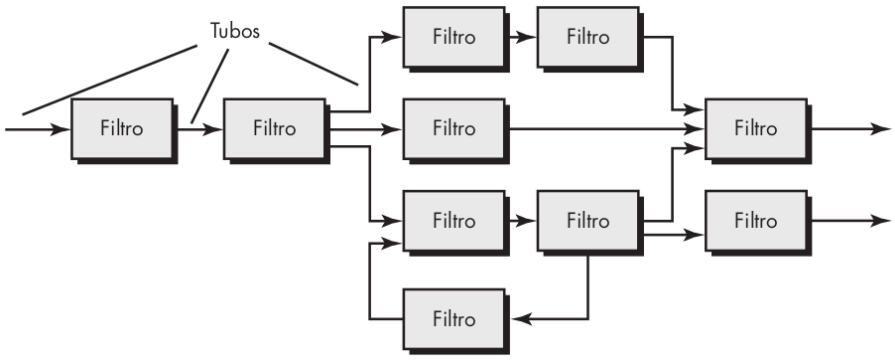


Figura 3: Arquitectura de flujo de datos con varios tubos y filtros, fuente [Pressman, 2010].

- "El principio de la sabiduría de un ingeniero de software es reconocer la diferencia entre conseguir que funcione un programa y hacerlo bien", Jackson
- El proceso de diseño implica la **selección de la mejor alternativa**
- Un buen diseño está formado por decisiones justificadas
- Se tiene que poder fácilmente recorrer las etapas (*traceability*: marcar una traza, seguir un rastro) o sea que **un requisito debe estar expresado en el diseño de alguna forma**
- Las decisiones de diseño deben estar basadas en un requisito o es una necesidad o una conveniencia de diseño
- Aplicar la **reusabilidad** siempre que sea posible. Reusar módulos/paquetes/clases/funciones
- Debe ser **uniforme** definiendo normas de estilo y formato a cumplir por los implicados
- Debe ser **integrado**, definiendo claramente el acoplamiento entre módulos
- **Diseñar no es escribir código** y escribir código no es diseñar. El nivel de abstracción del diseño es diferente al de la codificación y las decisiones de diseño no se deben hacer en la etapa de codificación; solamente se deben tomar decisiones de implementación [6].
- Se debe valorar la **calidad** del diseño mientras que se crea y no una vez finalizado
- Se debe estructurar para **admitir cambios**

[6] Esto entra en contradicción con lo que hemos visto sobre métodos iterativos e incrementales

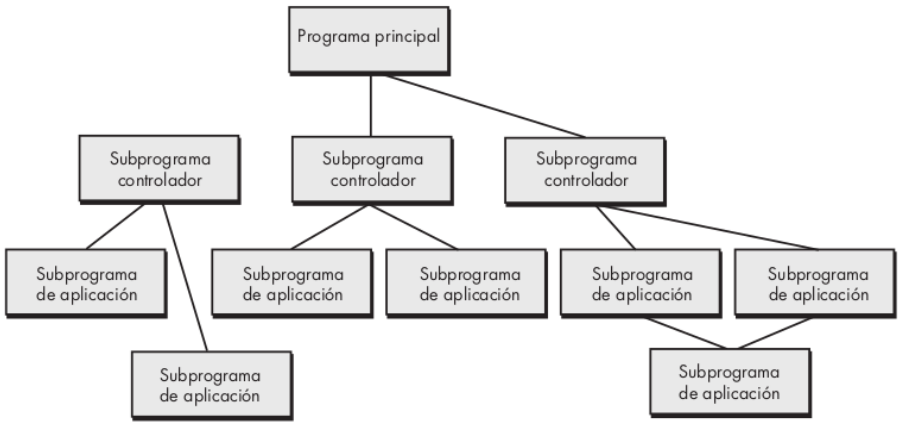


Figura 4: Arquitectura de programa principal/subprograma, fuente [Pressman, 2010].

- **Minimizar la distancia intelectual** entre el software y el problema
- Se debe **revisar** para eliminar errores

Además de estos principios, tenemos una lista similar en la entrada "Software Design"[7] en Wikipedia [Wikipedia, 2014d].

Conceptos de diseño

Algunos conceptos de diseño (muchos heredados de la orientación a objetos) que nos pueden ayudar en la definición de la arquitectura:

1. Abstracción
2. Refinamiento
3. Modularidad
4. Jerarquía de control
5. Particionamiento horizontal y vertical de la estructura
6. Estructuras de datos
7. Ocultamiento de información
8. Arquitectura y patrones de diseño (con propiedades y limitaciones conocidas)
9. ...

[7] http://en.wikipedia.org/wiki/Software_design#Software_Design

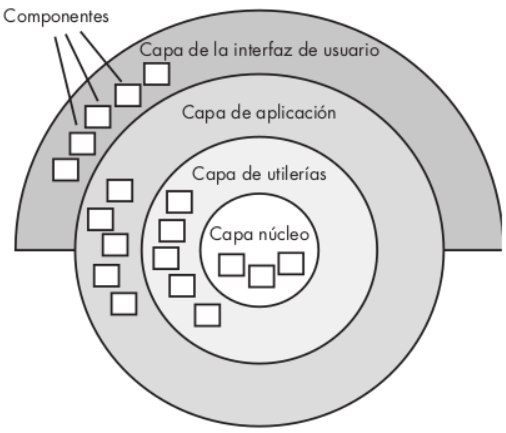


Figura 5: Arquitectura en capas, fuente [Pressman, 2010].

Consideraciones de diseño

Existen muchos aspectos que considerar cuando se diseña software. La importancia que se le de a cada aspecto dependerá de los objetivos que pretenda alcanzar el desarrollo software.

Algunos aspectos [Wikipedia, 2014d]:

- **Compatibilidad**. Capacidad de interoperar con otros productos o con otras versiones del mismo.
- **Extensibilidad**. Capacidad de añadir nuevas funcionalidades sin que impliquen grandes cambios en la arquitectura.
- **Tolerancia a fallos**. ¿Qué capacidad tiene el software para recuperarse de fallos en componentes?
- **Mantenibilidad**. Se refiere a la facilidad para arreglar errores o cambiar la funcionalidad. Una amplia mantenibilidad es el resultado de buenas extensibilidad y modularidad.
- **Modularidad**. El software se compone de componentes bien definidos e independientes. Esto mejora la implementación y prueba previa a la integración, y en consecuencia permite dividir/repartir el trabajo.
- **Fiabilidad**. El software es capaz de ejecutar una función bajo unas condiciones concretas durante un tiempo específico.
- **Seguridad**. El software resistirá actos e influencias hostiles.
- **Usabilidad**. El software debe ser usado por los usuarios finales (audiencia=.

- **Rendimiento**. El software realiza tareas en un tiempo aceptable y con un consumo de memoria no muy alto.
- **Portabilidad.** Se podrá utilizar en distintos entornos.
- **Escalabilidad.** El software se adapta bien a un aumento de datos, número de usuarios, etc.

Principios SOLID

Principios SOLID

Esta sección resume la clase especial *Cinco maneras de mejorar tu código: SOLID* impartida por Sergio Gómez. Los correspondientes códigos de ejemplo están colgados en Moodle.

Como veréis, en esta sección se conecta más la idea abstracta de buenas prácticas de diseño con el código. La idea general es conseguir **código de calidad**, es decir, reutilizable y extensible, y esto se apoya en buena parte en la idea de un buen reparto de responsabilidad y tratar de minimizar las *dependencias* entre clases (también llamado *acoplamiento*). En esta sección, además, usaremos información de los artículos de SOLID en inglés [Wikipedia, 2014b] y castellano [Wikipedia, 2014c] de Wikipedia.

Principios de diseño

Existen varios principios de diseño software para hacer código de calidad:

- *Don't Repeat Yourself* (**DRY**), no te repitas
- *You ain't gonna need it* (**YAGNI**), no lo hagas hasta que no lo necesites
- *Keep it simple, stupid* (**KISS**), mantenlo simple, estúpido
- **SOLID**:
  - **S**ingle responsibility principle, Principio de responsabilidad única
  - **O**pen/closed principle, Principio abierto/cerrado
  - **L**iskov substitution principle, Principio de sustitución de Liskov
  - **I**nterface segregation principle, Principio de segregación de la interfaz
  - **D**ependency inversion principle, Principio de inversión de dependencia

Principio de responsabilidad única

Principio de responsabilidad única
Principio de responsabilidad única (*Single responsibility principle*) dice que un objeto solo debería tener una única responsabilidad.

Ejemplo
En el ejemplo tenemos una clase *Libro* cuya responsabilidad es representar un libro. Ubicar en esta clase métodos para obtener un localizador de un libro o almacenarlo en la base de datos sería erróneo. Lo adecuado sería crear dos clases que se encargasen de esto.

Principio abierto/cerrado

Principio abierto/cerrado
"Las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación". Esto significa que el código debe estar abierto a incorporar nueva funcionalidad o mejoras, pero que estas mejoras no deberían alterar el código existen (salvo corrección de errores, obviamente).

Ejemplo
En el ejemplo de código, la extensión de funcionalidad se realiza incrementando el número de casos en una estructura de control *swicth*. Una solución más adecuada sería utilizar una clase abstracta y que la extensión de funcionalidad se haga con nuevas clases que implementen la clase abstracta, de esta forma la extensión se realiza sin modificar el código anterior.

Principio de sustitución de Liskov

Principio de sustitución de Liskov
Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa. Esto se traduce en que las clases hijas no deben modificar el comportamiento de las clases madre.

Ejemplo
En el ejemplo tenemos la clase *Rectangulo*, de la cuál hereda *Cuadrado* modificando los métodos *set* para que el ancho y alto sean iguales. El problema surge con la función que calcula el área, ya que que el comportamiento esperado de esta función es distinto para las dos clases. La clase hija debería poder utilizarse como la clase madre de manera idéntica sin tener que conocer la diferencia entre las clases. En el ejemplo el test de unidad fallará cuando no debería hacerlo.

Principio de segregación de la interfaz

Principio de segregación de la interfaz
Al plantear qué métodos debe definir una interfaz (en el sentido de clase abstracta o similares) no se debería establecer un conjunto único, sino que los métodos deberían agruparse por funcionalidad resultando en distintas interfaces.

Ejemplo
Por ejemplo, una interfaz para manejar una impresora no debería asumir que todas las impresoras tendrán un escáner o un fax integrado, de otra forma la interfaz obligaría a definir los métodos relacionados con escanear documentos a todas las clases que implementen la interfaz, aunque la impresora para la que se desarrolla el controlador no disponga de interfaz. Como alternativa las funcionalidades relacionadas con imprimir, escanear o enviar faxes deberían agruparse por separado.

Principio de inversión de dependencia I

Principio de inversión de dependencia
Este principio se refiere a una forma de desacoplar módulos software en el sentido de que la tradicional dependencia de los módulos de alto nivel respecto a los inferiores debe *invertirse* para romper esta dependencia de los detalles de implementación de bajo nivel. El principio dice:

1. Un módulo de alto nivel no debe depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
2. Las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones.

Principio de inversión de dependencia II

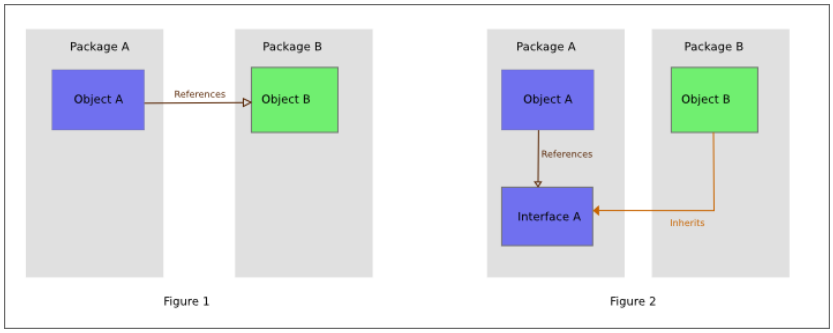


Figura 6: Ejemplo de inversión de dependencia, fuente [Wikipedia, 2014a].

La figura muestra un código con la misma funcionalidad, sin embargo, en la figura de la derecha se ha utilizado una interfaz para invertir la dependencia. **El patrón de diseño *injección de dependencia*** ayuda a cumplir este principio.

Ejemplo de rediseño: antes de SOLID

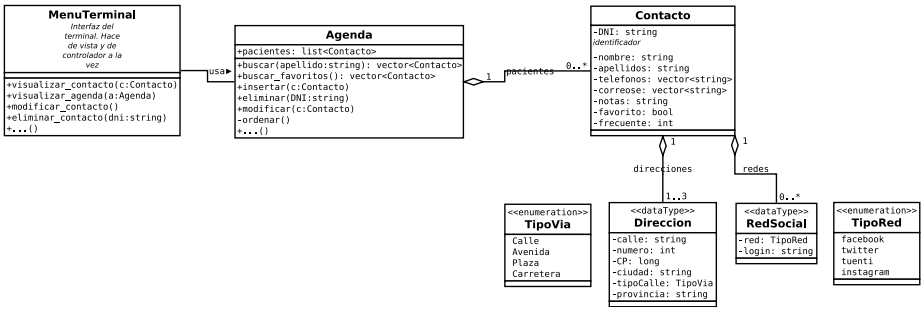


Figura 7: Diagrama de clases de la práctica 2 de la asignatura.

Ejemplo de rediseño: después de SOLID

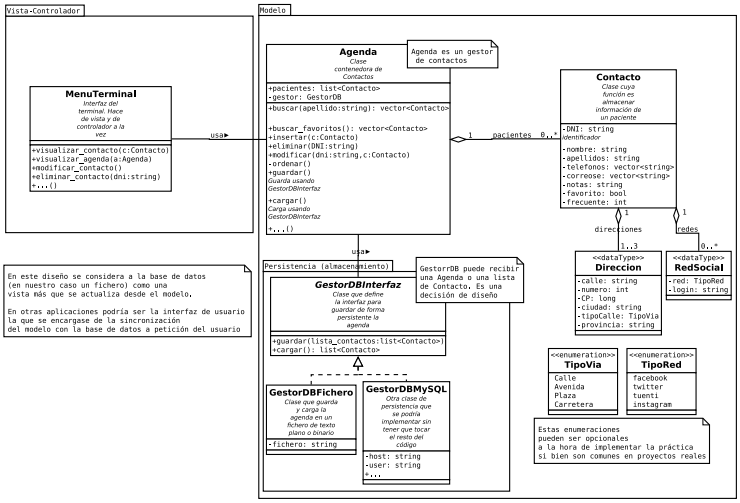


Figura 8: Diagrama de clases de la práctica 2 de la asignatura rediseñado según principios SOLID

4. Patrones de arquitectura y diseño software

4.1. Patrones de arquitectura y diseño

¿Qué es un patrón de diseño software?

- Los **patrones de diseño software** (*software design patterns*) son soluciones generables y reutilizables a problemas comunes en el diseño de software.

- Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su **efectividad** resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser **reutilizable**, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.
- Por tanto, **un patrón no es una solución en si**, sino una plantilla o descripción para desarrollar una solución.

Niveles en los patrones

Aunque en algunos textos sólo leeréis hablar de *patrones de diseño* (nivel de módulo y código), también se debe hablar de *patrones de arquitectura* cuando se refieren a la estructuración del código a un nivel superior de abstracción.

También encontraremos patrones de diseño para **dominios de problemas concretos**: programación concurrente y paralela, programación distribuida, programación de interfaces de usuario, etc.

Ventajas del uso de patrones

Según Pradel i Miquel and Martos [2010] algunas ventajas son:

- Reutilizar las soluciones y aprovechar la experiencia previa de otras personas que han dedicado más esfuerzo a entender los contextos, las soluciones y las consecuencias del que nosotros queremos o podemos dedicar.
- Beneficiarnos del conocimiento y la experiencia de estas personas mediante un enfoque metódico.
- Comunicar y transmitir nuestra experiencia a otras personas (si definimos nuevos patrones).
- Establecer un vocabulario común para mejorar la comunicación.
- Encapsular conocimiento detallado sobre un tipo de problema y sus soluciones asignándole un nombre con la finalidad de poder hacer referencia a éstos fácilmente.
- No tener que reinventar una solución al problema.

Precauciones en el uso de patrones

Según Pradel i Miquel and Martos [2010] algunos posibles inconvenientes son:

- Los patrones no evitan que tengamos que razonar sobre problemas de desarrollo.

- No hay que asumir que utilizar un patrón nos proporciona la mejor solución posible.
- Al seleccionar un patrón se debe asegurar que se entiende su funcionamiento correctamente, y razonar cuáles son las consecuencias (beneficios y limitaciones) de aplicarlo a nuestro problema.

Breve reseña histórica

- Los patrones en el diseño en existen desde hace años en artesanía, costura, ciencia e ingenierías.
- En los 80 empezó a hablar de patrones en el campo de la informática. La primera publicación fue *Using Pattern Languages for OO Programs* de Ward Cunningham y Kent Beck.
- La popularidad de los patrones de diseño software vino de la mano del trabajo de Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides en el libro *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma et al., 2003].
- El libro propuso una lista inicial de patrones clasificados en tres categorías: **patrones de creación**, **patrones de estructura** y **patrones de comportamiento**.
- A los autores del libro se les conoce como ***The "Gang of Four"* (GoF)**, y es por esto que al conjunto inicial de patrones de diseño se les conoce como patrones GoF o *GoF patterns*.
- El libro del GoF presenta un catálogo inicial de patrones y además un ejemplo de cómo desarrollar un editor de texto WYSIWYG (*What You See Is What You Get*, "lo que ves es lo que obtienes")[8].

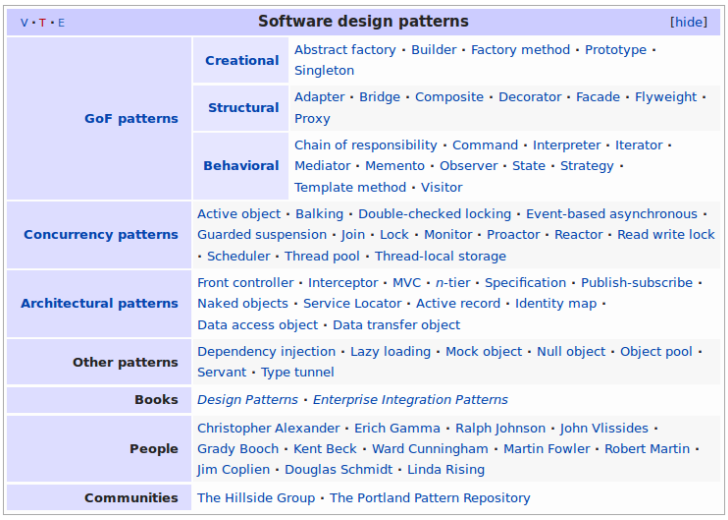
Clasificación de patrones de diseño software

Clasificación inicial del GoF [Gamma et al., 2003]:

- Los **patrones de diseño de creación** abstraen el proceso de creación de instancias. Ayudan a hacer un sistema independiente de cómo se crean, se componen y se representan sus objetos. Ejemplo: *factoría abstracta* y *singleton*.
- Los **patrones de diseño estructurales** se ocupan de cómo se combinan las clases y los objetos para formar estructuras más grandes. Ejemplo: *fachada*.

[8] http://es.wikipedia.org/wiki/WYSIWYG

- Los **patrones de diseño de comportamiento** tienen que ver con algoritmos y con asignación de responsabilidades a objetos. Estos patrones no describen sólo patrones de clases y objetos, sino también patrones de comunicación entre ellos. Ejemplos: *iterador* y *observador*.



V • T • E	Software design patterns	[hide]
GoF patterns	Creational	Abstract factory • Builder • Factory method • Prototype • Singleton
	Structural	Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy
	Behavioral	Chain of responsibility • Command • Interpreter • Iterator • Mediator • Memento • Observer • State • Strategy • Template method • Visitor
Concurrency patterns		Active object • Balking • Double-checked locking • Event-based asynchronous • Guarded suspension • Join • Lock • Monitor • Proactor • Reactor • Read write lock • Scheduler • Thread pool • Thread-local storage
Architectural patterns		Front controller • Interceptor • MVC • *n*-tier • Specification • Publish-subscribe • Naked objects • Service Locator • Active record • Identity map • Data access object • Data transfer object
Other patterns		Dependency injection • Lazy loading • Mock object • Null object • Object pool • Servant • Type tunnel
Books		*Design Patterns* • *Enterprise Integration Patterns*
People		Christopher Alexander • Erich Gamma • Ralph Johnson • John Vlissides • Grady Booch • Kent Beck • Ward Cunningham • Martin Fowler • Robert Martin • Jim Coplien • Douglas Schmidt • Linda Rising
Communities		The Hillside Group • The Portland Pattern Repository

Figura 9: Clasificación de patrones de diseño software según Wikipedia.

Estructura de un patrón

Los elementos de la estructura de un patrón varían según los autores, si bien hay un conjunto de elementos mínimos comunes a la mayoría de ellos.

Elementos fundamentales

Los cinco elementos fundamentales que tiene que tener todo patrón son el nombre, el contexto, el problema, la solución y las consecuencias de aplicarlo.

Relación entre principios y patrones de diseño

Los **principios de diseño son unas normas generales** que nos ayudan a crear software de mayor calidad, reusabilidad y extensibilidad. Esto es uno de los principales objetivos de la ingeniería del software. Además, en el tema de pruebas software, veremos cómo las clases y métodos que cumplan los principios de diseño serán más fáciles de probar al reducir su *complejidad ciclomática*.

Los **patrones de diseño son propuestas de soluciones** a problemas comunes que en general nos permitirán satisfacer varios principios de diseño. Así pues no sólo nos aportan un borrador de solución a un problema, sino que **la solución cumplirá muchos de los principios de diseño**.

4.2. Ejemplos de patrones

Patrones de arquitectura y de diseño

Los **patrones de arquitectura** definen la organización de una aplicación a grandes rasgos. Los **patrones de diseño** se refieren a niveles menos abstractos. A menudo, patrones de estos dos niveles estarán relacionados. Por ejemplo, veremos cómo usar el patrón *Observer* es esencial dentro de la arquitectura modelo-vista-controlador.

La los patrones de ejemplo de esta sección están sacado del libro original del GoF [Gamma et al., 2003]. Aquí sólo se presenta un resumen de lo explicado en clase.

Patrón único o *singleton*

El **patrón de creación único** o *singleton* garantiza que sólo exista una instancia de una clase, y proporciona un punto de acceso global a ella.

Ejemplos de motivación

Aunque puedan existir varias impresoras, es importante que sólo exista una cola de impresión. La configuración de un programa debería guardarse en una única instancia a la que accedan directamente todos los componentes de programa. Una variable global puede hacer accesible un objeto, pero no evita que haya varias instancias de este objeto. El patrón *único* transfiere la responsabilidad de garantizar que sólo exista una instancia a la propia clase.

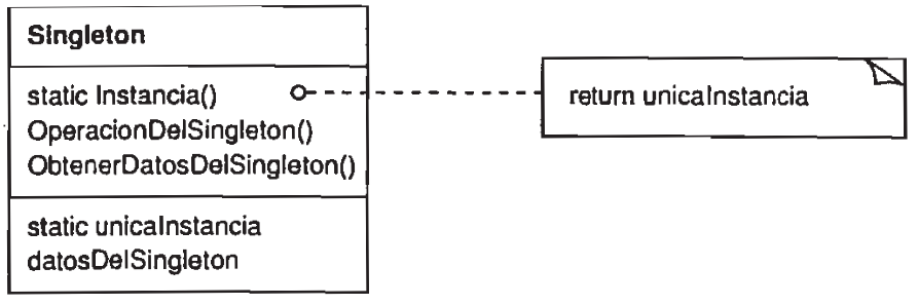


Figura 10: Estructura del patrón único, fuente [Gamma et al., 2003]

Patrón factoría abstracta

El patrón **factoría abstracta** (*abstract factory*) es un **patrón de creación** que proporciona una interfaz para crear familias de objetos relacionados o que dependen entre si sin especificar sus clases concretas.

Ejemplo de motivación
Pensemos en desarrollar una aplicación de escritorio que queremos que funciona con distintas bibliotecas de construcción de interfaces gráficas, de formas que sea fácil portar la aplicación a otras plataformas o incluso que tenga diferente aspecto o comportamiento dentro de una misma plataforma. Para poder realizar esto no se deberían codificar los elementos gráficos de esta que interaccionan con el usuario o muestran información pensando en una implementación concreta, ya que sería más difícil cambiar de motor de interfaces más tarde. El patrón *factoría abstracta* nos proporciona un diseño que nos permite desarrollar una aplicación que supere estas limitaciones.

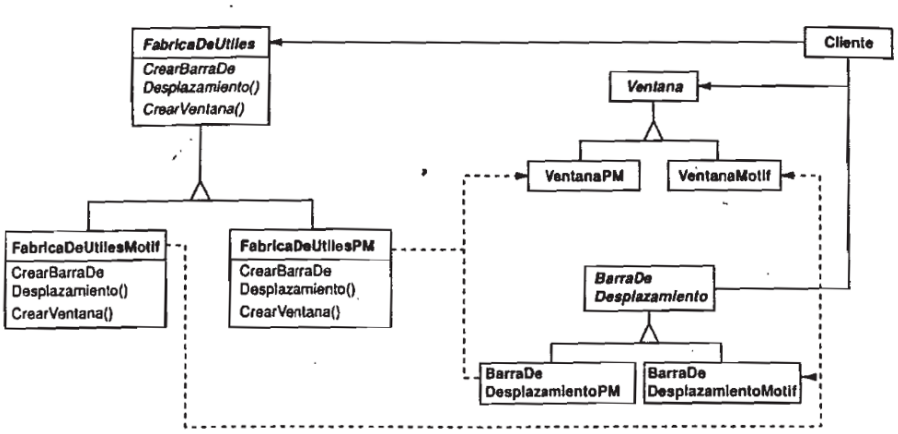


Figura 11: Ejemplo de diagrama de clases del patrón factoría abstracta, fuente [Gamma et al., 2003]

Patrón iterador

El patrón **iterador** (*iterator*) es un **patrón de comportamiento** que proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

Ejemplo de motivación
Con objetos agregados como las *listas*, normalmente es preferible poder acceder a sus elementos sin exponer su estructura (implementación) interna. Además, normalmente con objetos de este tipo se desea poder recorrerlos de diferentes formas. Si quisiéramos contemplar las distintas formas de recorrer una *lista* necesitaríamos modificar la interfaz de ésta con distintas operaciones según distintos tipos de recorridos. Por otro lado, se puede dar el caso de que se quiera realizar más de un recorrido simultáneamente sobre la lista, lo que complicaría o incluso haría imposible proporcionar esta funcionalidad.

El patrón **iterador** nos permite realizar estas operaciones trasladando la responsabilidad de acceder y recorrer el objeto *lista* a un objeto **iterador**. Esta clase

definirá la interfaz de acceso a los elementos de la *lista* y será responsable de saber cuál es el elemento actual y cuáles se han recorrido ya.

Las tres siguientes figuras nos muestran el proceso de entender el diseño del patrón iterador.

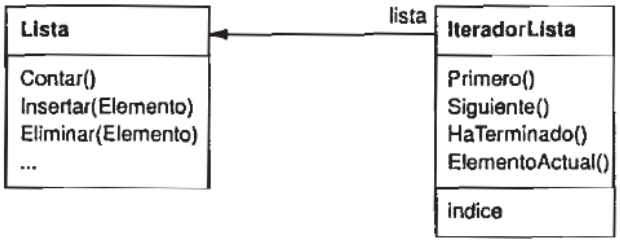


Figura 12: Primer diseño del patrón iterador, fuente [Gamma et al., 2003].

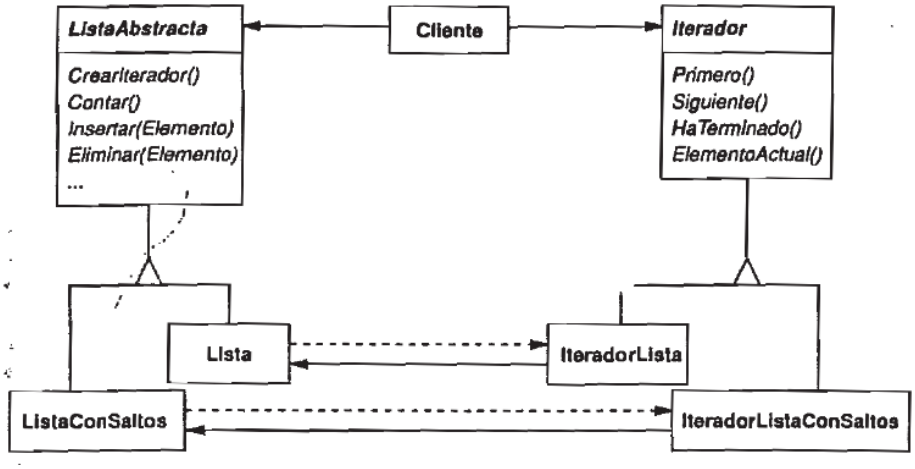


Figura 13: Primer diseño del patrón iterador, fuente [Gamma et al., 2003].

Patrón observador

El patrón **observador** u **observer** es un **patrón de comportamiento** que define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifiquen y actualicen automáticamente todos los objetos que depende de él. También es conocido por el nombre *dependiente* (*Dependents*) y *Publicador-Subscriptor* (*Publish-Subscribe*).

Ejemplo de motivación

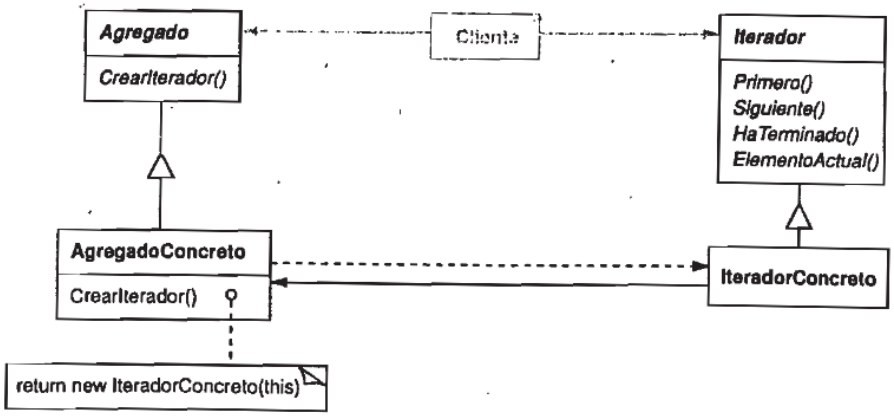


Figura 14: Primer diseño del patrón iterador, fuente [Gamma et al., 2003].

Supongamos una hoja de cálculo con una información en base a la cuál se actualizan distintas gráficas, incluyendo una representación en formato de tabla. Es deseable que la hoja de cálculo esté desacoplada de los distintos tipos de gráficos, de forma que la hoja que contiene los datos es independiente de los gráficos que se generan a través de ella. Sin embargo, desde el punto de vista de usuario se desea que cualquier cambio en los datos se refleje inmediatamente en las gráficas. Cada una de las representaciones de los datos sería un objeto observador sobre los datos.

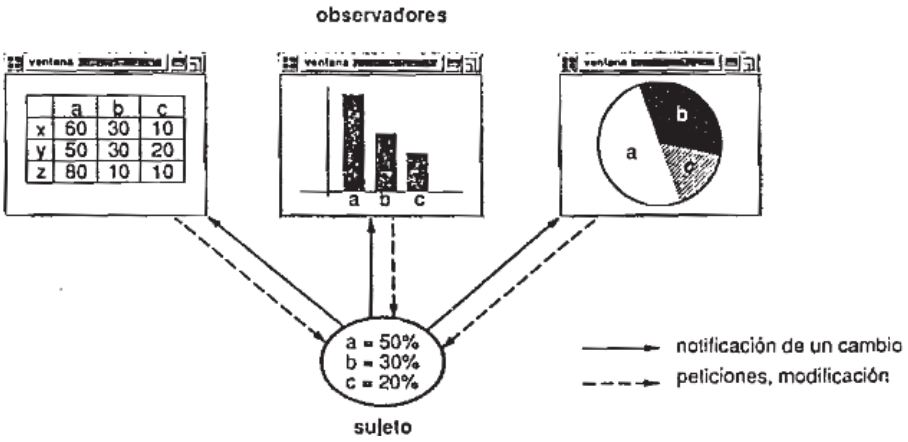


Figura 15: Ejemplo de patrón observador sobre una hoja de cálculo, fuente [Gamma et al., 2003].

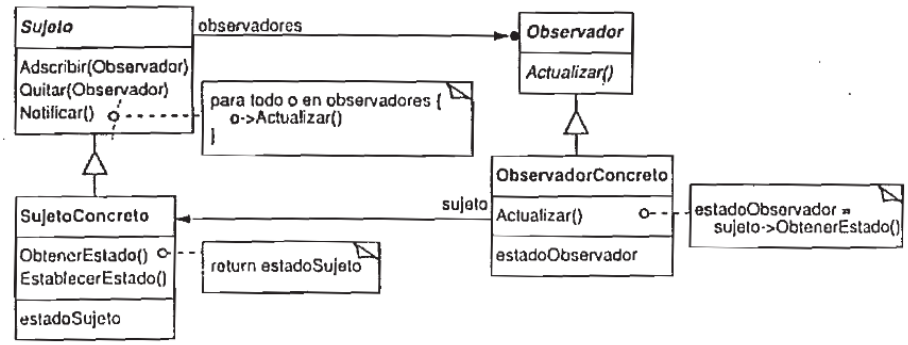


Figura 16: Diagrama de clases del patrón observador, fuente [Gamma et al., 2003].

Patrón modelo-vista-controlador

Modelo-vista-controlador

El **patrón de arquitectura modelo-vista-controlador** (*Model-View-Controller*, MVC) divide una aplicación interactiva en tres componentes para poder separar los datos y la lógica del problema de la interfaz de usuario y de los eventos y comunicaciones.

Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento Wikipedia [2014e].

Componentes

A grandes rasgos: el **modelo** contiene la funcionalidad principal y los datos; las **vistas** muestran información al usuario; los **controladores** se encargan de las entradas del usuario. Las vistas y los controladores en conjunto definen la interfaz de usuario y un mecanismo de propagación de cambios se encarga de asegurar la consistencia entre la interfaz de usuario y el modelo.

Ejemplo MVC

Motivación al uso de MVC

En **aplicaciones interactivas con interfaz de usuario** será relativamente común:

- Las interfaces de usuario tienden a recibir peticiones de cambio. Por ejemplo al incorporar nueva funcionalidad se debe incorporar esta a los menús.
- Un sistema desea ser *portado* a otra plataforma

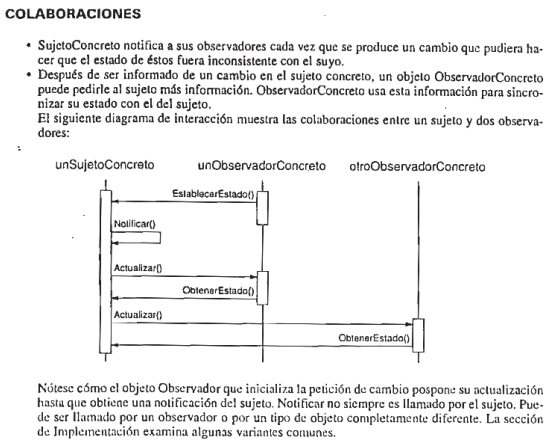


Figura 17: Colaboración entre elementos del patrón observador, fuente [Gamma et al., 2003].

- Distintos tipos de usuarios esperan distintos tipos de interfaces incluso con requisitos que entren en conflicto.

Para que sea **fácil** realizar este tipo de modificaciones en el software con la suficiente **flexibilidad** y **minimizando los errores**, **será necesario que el núcleo de la funcionalidad esté lo más desacoplado posible de la interfaz de usuario**. De lo contrario sería necesario mantener distintos sistemas simultáneamente, tantos como interfaces de usuario, cuando lo deseable sería que el núcleo de la aplicación no cambie si la lógica de negocio no cambia.

La solución
El MVC da solución a este problema al dividir la funcionalidad de una aplicación interactiva entre: procesamiento, salida y entrada.

Componentes
De manera genérica, los componentes de MVC se podrían definir como sigue Wikipedia [2014e]:

- El **Modelo**: Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (*lógica de negocio*). Envía a la **vista** o **vistas** aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al **modelo** a través del **controlador** o **controladores**.

- El **Controlador** (uno o varios): Responde a eventos (normalmente acciones del usuario) e invoca peticiones al **modelo** cuando se hace alguna

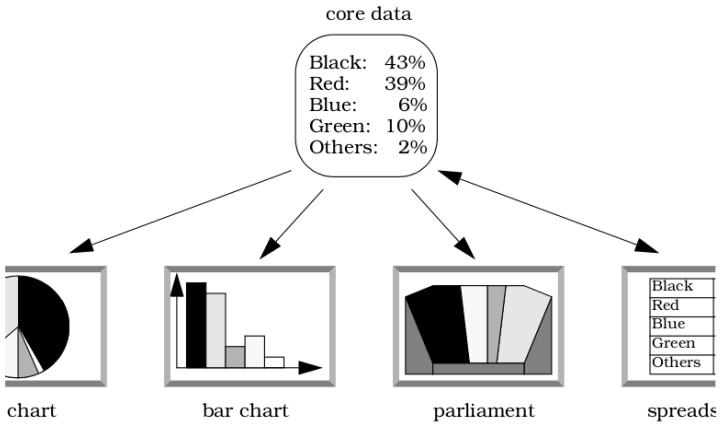


Figura 18: Ejemplo de aplicación con varias vistas siguiendo el patrón arquitectónico MVC.

solicitud sobre la información (por ejemplo, modificar un documento o un registro en una base de datos). También puede enviar comandos a su **vista** asociada si se solicita un cambio en la forma en que se presenta de **modelo** (por ejemplo, desplazamiento o *scroll* por un documento o por los diferentes registros de una base de datos), por tanto se podría decir que el **controlador** hace de intermediario entre la **vista** y el **modelo**.

- La **Vista** (una o varias): Presenta el **modelo** (información y lógica de negocio) en un formato adecuado para interactuar (normalmente la interfaz de usuario) por tanto requiere de dicho **modelo** la información que debe representar como salida.

Diagrama de clases MVC

5. Frameworks

Frameworks

Definición de *framework*

Framework
Un *framework* o *marco de trabajo* es un conjunto de clases predefinidas que tenemos que especializar y/o instanciar para implementar una aplicación o subsistema ahorrando tiempo y errores. El *framework* nos ofrece una funcionalidad genérica ya implementada y probada que nos permite centrarnos en lo específico de nuestra aplicación.

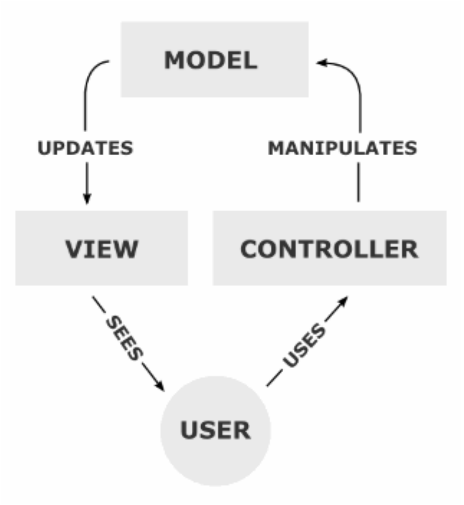


Figura 19: Colaboración entre componentes del MVC.

- No debemos confundir un *framework* con una biblioteca de clases. Un *framework* es un tipo especial de biblioteca que **define el diseño de la aplicación que lo utilizará**.
- Un *framework* **comparte el control de la ejecución** con las clases que lo utilizan siguiendo el "principio de Hollywood" (En inglés, "*Don't call us. We'll call you*" (no hace falta que nos llames, nosotros te llamaremos).): el *framework* implementará el comportamiento genérico del sistema y llamará a nuestras clases en aquellos puntos donde haya que definir un comportamiento específico.
- Como un *framework* tiene que llamar a nuestras clases, **definirá la interfaz que espera que tengan estas clases**. Así, para utilizar un *framework* tendremos que implementar el comportamiento específico de nuestra aplicación respetando las interfaces y colaboraciones definidas en él.

Ejemplo de conexión clases aplicación-framework

En este caso, por ejemplo, nuestra aplicación utiliza un *framework* mediante la herencia (de la clase Cc hacia C2) y la implementación de una interfaz (Ca implementa I1) que es utilizada por el *framework* (por C2 y C3). De esta manera, el *framework* llevará el control de la ejecución llamando a las clases Ca y Cc de nuestra aplicación cuando convenga.

Ventajas e inconvenientes

- Reutilizamos un diseño ahorrando tiempo y esfuerzo.

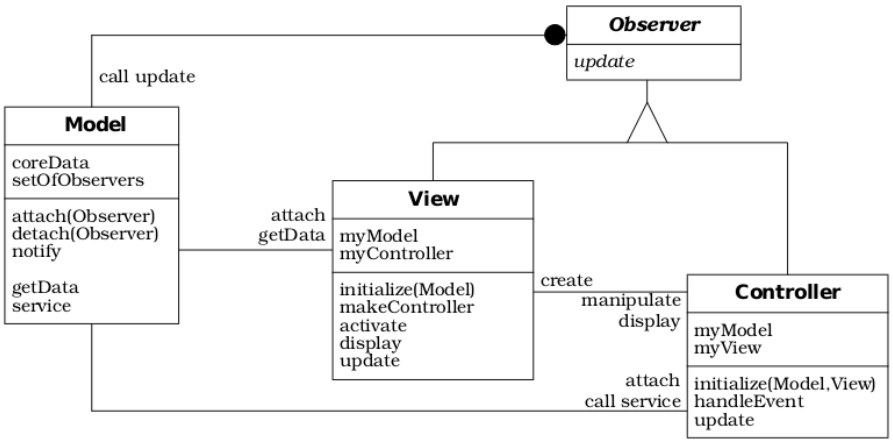


Figura 20: Ejemplo de diagrama de clases MVC junto con Observer.

- Dado que el control es compartido entre la aplicación y el propio *framework*, un *framework* puede contener mucha más funcionalidad que una biblioteca tradicional.
- Dan una implementación parcial del diseño que queremos reutilizar.

El principal inconveniente del uso de *framework* es la necesidad de un proceso inicial de aprendizaje. Para reutilizar el diseño del *framework*, primero lo tenemos que entender completamente.

Relación entre *frameworks* y patrones

- La principal diferencia entre un patrón y un *framework* es que el *framework* es un elemento de software, mientras que el patrón es un elemento de conocimiento sobre el software.
- El patrón debe implementarse desde cero adaptándose al caso concreto.
- En el *framework* extenderemos la funcionalidad de un conjunto de clases ya implementadas sin modificarlas.
- Los *framework* suelen hacer uso intensivo de patrones de diseño, de manera que conocer los patrones nos ayudará a entenderlos.

El framework dentro del proceso de desarrollo

El uso de *frameworks* afecta a nuestro proceso de desarrollo de software. Dado que estamos reutilizando un diseño previo, tenemos que acomodar nuestro sistema al diseño que queremos reutilizar.

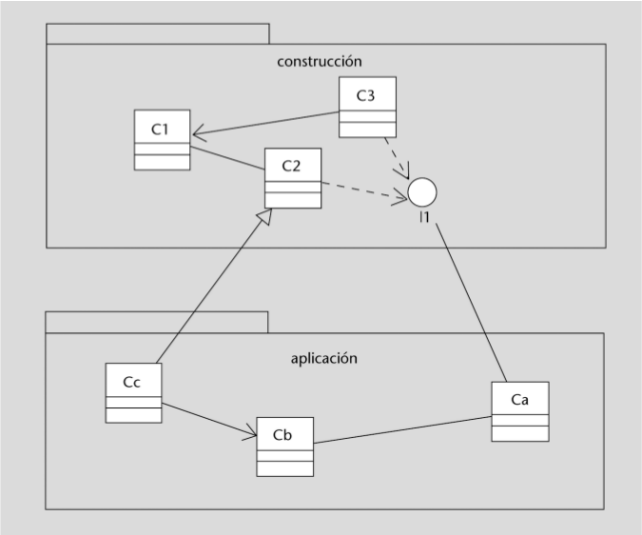


Figura 21: Relación entre las clases de un framework y de la aplicación, fuente [Pradel i Miquel and Martos, 2010].

6. Bibliografía

Fuentes

El contenido de estos apuntes se basa a su vez en otros apuntes (como Pradel i Miquel and Martos [2010]) y en los libros de Pressman [2010] y Gamma et al. [2003], así como en las referencias citadas.

Bibliografía y enlaces

Referencias

E. Freeman, E. Robson, B. Bates, and K. Sierra. *Head First Design Patterns*. O'Reilly Media, 2004. ISBN 978-0-596-00712-6.

E. Gamma, R. Helm, R. Johnson, and J. Vlisides. *Patrones de Diseño. Elementos de software orientado a objetos reutilizable*. Pearson Education. Addison-Wesley, 2003. ISBN 84-7829-059-1.

L. R. Irene T. Apuntes de la Asignatura Ingeniería del Software. Tema 6: Introducción al diseño. 2013.

Microsoft. Microsoft application architecture guide, 2014. URL http://msdn.microsoft.com/en-us/library/ee658117.aspx. Consultado en diciembre de 2014.

Object Management Group. Data Distribution Service Portal, Noviembre 2014. URL http://portals.omg.org/dds/.

J. Pradel i Miquel and J. A. R. Martos. *Introducción a los patrones. Asignatura de Análisis y diseño con patrones*. Universitat Oberta de Catalunya, 2010. FUOC PID_00165657.

R. Pressman. *Ingeniería del Software*. McGraw-Hill, 7ª edición edition, 2010. ISBN 9786071503145.

P. Ralph and Y. Wand. A Proposal for a Formal Definition of the Design Concept. In K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson, editors, *Design Requirements Engineering: A Ten-Year Perspective*, volume 14 of *Lecture Notes in Business Information Processing*, pages 103–136. Springer Berlin Heidelberg, 2009. ISBN 978-3-540-92965-9. doi: 10.1007/978-3-540-92966-6_6. URL http://dx.doi.org/10.1007/978-3-540-92966-6_6.

Wikipedia. Dependency injection, 2014a. URL http://en.wikipedia.org/wiki/Dependency_injection. [Online; accessed 18-December-2014].

Wikipedia. SOLID (inglés), 2014b. URL http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29. [Online; accessed 18-December-2014].

Wikipedia. SOLID (castellano), 2014c. URL http://es.wikipedia.org/wiki/SOLID_%28object-oriented_design%29. [Online; accessed 18-December-2014].

Wikipedia. Software design — wikipedia, the free encyclopedia, 2014d. URL http://en.wikipedia.org/w/index.php?title=Software_design&oldid=635193277. [Online; accessed 25-November-2014].

Wikipedia. Modelo–vista–controlador, 2014e. URL http://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador. [Online; accessed 18-December-2014].