Desarrollo de Sistemas Software

DSS

2. Introducción al diseño de software

Diseño de software

¿Qué es el diseño de software?
Es una disciplina de la Ingeniería de Software

- Análisis, diseño e implementación son disciplinas centrales de la ingeniería del software
- Están fuertemente relacionadas, y las decisiones sobre cómo realizar una de ellas afecta directamente a las demás
- Lenguajes, herramientas, actividades a realizar, etc. deben decidirse al comienzo de un proyecto.

Diseño explícito: realizado para planificar o documentar el software desarrollado
Diseño implícito: la estructura que el software tiene realmente, aunque no se haya diseñado formalmente.

Fases del diseño

- **Diseño arquitectural:** estructura de alto nivel, identificación de los componentes principales junto con los requisitos funcionales y no funcionales
- **Diseño detallado:** los componentes se descomponen con un nivel de detalle más fino. Guiado por la arquitectura y los requisitos funcionales

Ventajas de un buen diseño

Aumenta las probabilidades de finalizar con éxito el proyecto.

- Ayuda a trasladar las especificaciones a los programadores de forma no ambigua.
- Ayuda a escribir código de calidad:
    - … conforme a las especificaciones
    - … fácil de mantener y extender
    - … que los demás pueden comprender

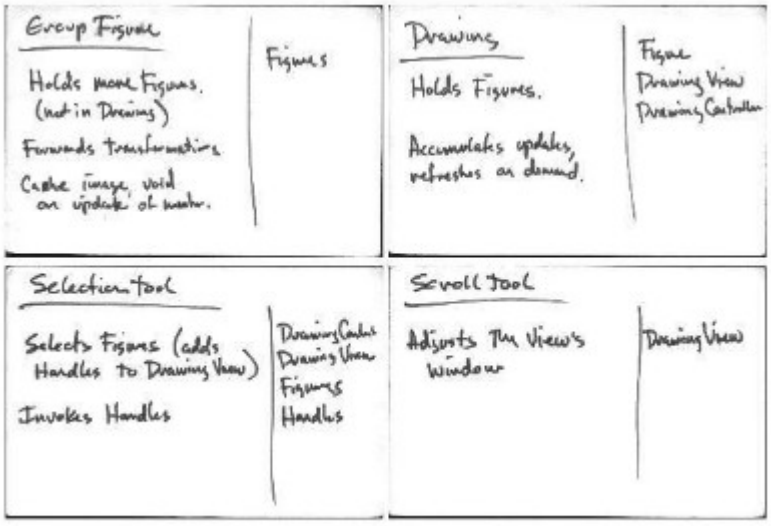
Diseñar bien no es garantía de éxito

Realizar un diseño previo no garantiza que el software resultante sea conforme a ese diseño ni a la especificación.
Existen disciplinas y técnicas para garantizar la calidad del software (p. ej. pruebas de código)

¿Cómo y cuánto hay que diseñar?

- **Diseño orientado a objetos:** los sistemas de software se componen de objetos con un estado privado y que interactúan entre sí
- **Diseño orientado a funciones:** descompone el sistema en un conjunto de funciones que interactúan y comparten un estado centralizado
- **Diseño de sistemas de tiempo real:** reparto de responsabilidades entre software y hardware para conseguir una respuesta rápida.
- **Diseño de interfaces de usuario:** tiene en cuenta las necesidades, experiencia y capacidades de los usuarios

Tarjetas CRC



Herramientas de diseño orientado a objetos

UML, diagramas más usados según Sommerville (2010)

- **Diagramas de actividad:** actividades involucradas en un proceso o en el procesamiento de datos
- **Diagramas de caso de uso:** interacciones entre el sistema y su entorno
- **Diagramas de secuencia:** interacciones entre los actores y el sistema y entre componentes del sistema
- **Diagramas de clases:** clases de objetos y sus asociaciones
- **Diagramas de estados:** cómo reacciona el sistema ante eventos internos y externos

UML no es más que un formato de representación.
Usar UML no garantiza que el diseño sea bueno

Formas de usar UML

- Como un medio para facilitar el debate sobre un sistema existente o propuesto
- Como una forma de documentar un sistema existente
- Como una descripción detallada que se puede usar para desarrollar una implementación
- Usando herramientas especializadas, los modelos se pueden usar para generar código automáticamente (Model-Driven Engineering, MDD)

La metodología impone las reglas

En metodologías tradicionales hay que generar numerosos modelos antes de tocar una sola línea de código.
Las metodologías ágiles recomiendan modelar lo mínimo imprescindible, en grupo y de manera informal -> los modelos se usan como herramienta de comunicación.

3. Patrones arquitecturales - Arquitectura en capas

Distintas definiciones

- “Descomposición de un sistema en componentes de alto nivel”
- “Decisiones difíciles de cambiar”
- “La arquitectura es el puente entre los objetivos de negocio (a menudo abstractos) y el sistema resultante (concreto)”

Un diseño arquitectural es una abstracción que muestra únicamente los detalles relevantes. La definición de una arquitectura debe incluir también una descripción del comportamiento de sus componentes. La arquitectura prescribe la manera en la que se deben implementar los componentes y cómo deben interactuar, normalmente mediante restricciones. Se deben tener en cuenta los requisitos no funcionales (p. ej. rendimiento esperado) y la distribución lógica y física de sus componentes. Las aplicaciones que no tienen una arquitectura formal, normalmente son:

- Altamente acopladas
- Frágiles
- Difíciles de cambiar
- No tienen una visión o dirección claras
- Es muy difícil determinar las características arquitecturales de una aplicación sin comprender el funcionamiento interno de cada componente y módulo en el sistema

Hay muchas formas de diseñar mal, pero sólo unas pocas de hacerlo bien y el éxito en el diseño arquitectural es complejo y desafiante, por eso los diseñadores han encontrado maneras de capturar y reutilizar el conocimiento adquirido en otros sistemas. Los patrones arquitecturales son formas de capturar estructuras de diseño de eficacia probada, de manera que puedan ser reutilizadas.

¿Qué es un patrón de diseño?

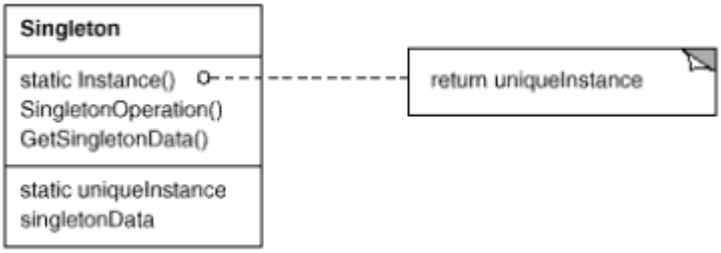
La mayoría de problemas de diseño no son nuevos, sólo cambia el dominio de la aplicación. No sólo se reutiliza el código, también podemos reutilizar soluciones anteriores. Podemos reutilizar la experiencia de los expertos documentada a través de patrones de diseño. Son soluciones documentadas a problemas frecuentes en el diseño de software:

- Nombre del patrón
- Problema y su contexto
- Solución
- Consecuencias

Conocer los patrones de diseño más habituales ayuda a comprender mejor los sistemas, incluso observando directamente el código.

Patrón Singleton

- Motivación: asegurar que una clase sólo tiene una única instancia, y proporcionar un punto de acceso global.
- Solución:



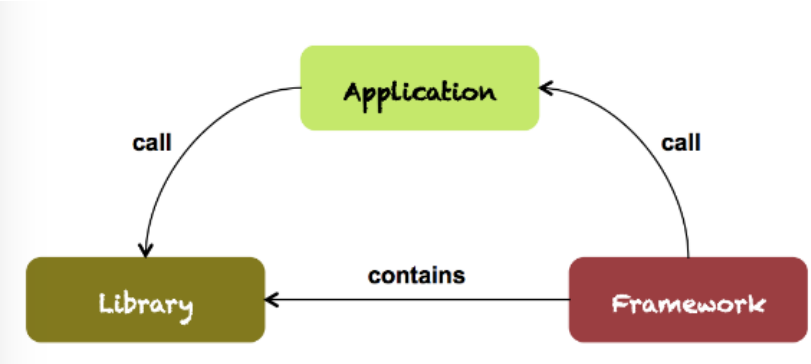
- Consecuencias: acceso controlado a la única instancia, mejor que el uso de variables globales

Los patrones arquitecturales expresan esquemas de organización estructural fundamentales, proporcionan un conjunto de subsistemas predefinidos, especifican sus responsabilidades e incluyen reglas y principios para organizar sus relaciones. Para los patrones más habituales es frecuente encontrar **framework** que permiten implementar una aplicación sin tener que escribirla desde cero.

Frameworks

Un framework es una aplicación reutilizable, "semi-completa" que se puede especializar para producir aplicaciones personalizadas. Incorporan numerosos patrones de diseño en su implementación.
Muchos frameworks siguen el principio **Convention over configuration:** no es necesario crear archivos de configuración si se respetan las convenciones de nombres (p. ej. mapeado de rutas en ASP.NET MVC)

- **Modularidad:** encapsulando los detalles de implementación detrás de interfaces estables
- **Reusabilidad:** definiendo componentes genéricos que se pueden reutilizar para crear nuevas aplicaciones
- **Extensibilidad:** proporcionando métodos "gancho" que permiten a las aplicaciones extender sus interfaces estables
- **Inversión de control** (Principio Hollywood): permite al framework (en lugar de a cada aplicación) determinar qué métodos de la aplicación serán invocados como respuesta a eventos externos.



Beneficios de usar frameworks:

- Desarrollo más rápido

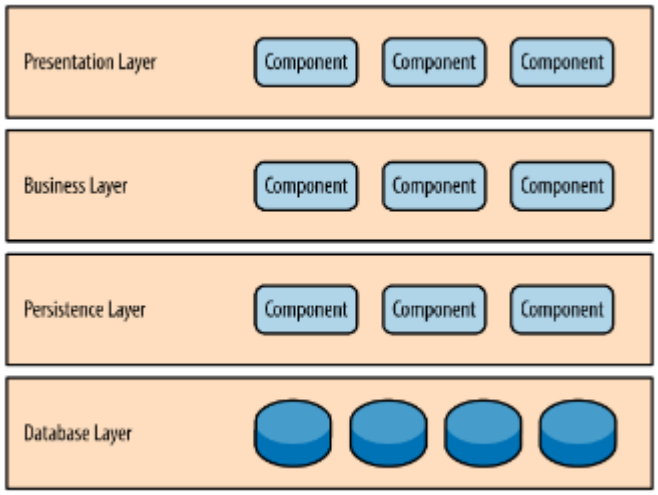
- Código más estable y robusto
- Mayor seguridad
- Menor coste de desarrollo
- Soporte de la comunidad

Desventajas del uso de frameworks:

- Curva de aprendizaje
- Problemas de integración con otros frameworks o librerías
- Limitaciones
- Es más difícil depurar el código
- El código es público

Arquitectura en capas

Descompone una aplicación en componentes situados en distintos niveles horizontales llamados capas



Cada capa se encarga de una tarea específica, abstrayendo los detalles a las demás capas -> separación de intereses. Capas típicas (aunque el patrón no prescribe ninguna):

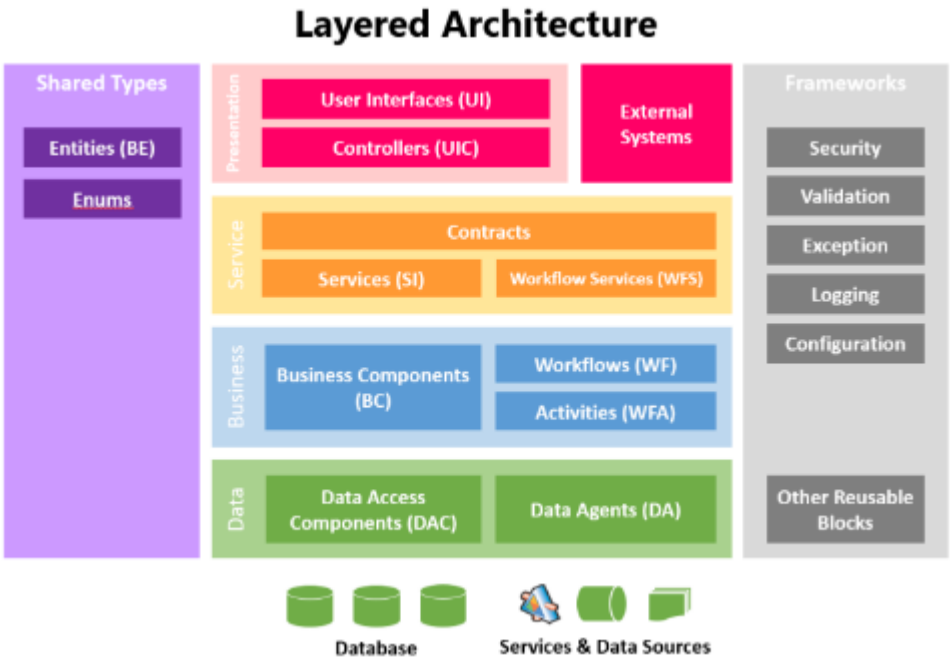
- **Presentación**: interacción con el usuario
- **Servicios**: funcionalidades de alto nivel
- **Lógica** de negocio: ejecución de las reglas de negocio
- **Persistencia (acceso a datos)**: comunicación con la BBDD
- **Base de datos**: almacenamiento de información

La capa de lógica de negocio y la capa de persistencia no deben depender nunca de la capa de presentación. La capa de lógica de negocio y la capa de persistencia pueden juntarse en una única capa en algunas circunstancias. La capa de servicios ofrece funcionalidades compuestas a partir de las clases de la capa de lógica de negocio, ocultando su complejidad.

- **Arquitectura cerrada:** la comunicación va de una capa a la inmediatamente inferior. Disminuye el impacto de los cambios y la complejidad del sistema.
- **Arquitectura abierta:** las capas superiores se pueden comunicar con todas o algunas de las inferiores. Necesario cuando las muchas peticiones se limitan a pasar de una capa a otra sin ninguna lógica de negocio asociada.

Se pueden crear tantas capas como sea necesario, siempre que cada capa tenga un propósito claro y separado de las demás.

Ejemplo: arquitectura .NET



- Es la arquitectura más adecuada para la mayoría de aplicaciones.
- No es necesaria cuando todas las peticiones pasan de una capa a otra sin procesamiento adicional (antipatrón sumidero)
- Cuando las capas no están distribuidas físicamente puede terminar convirtiéndose en una aplicación monolítica, lo que puede ser un riesgo para la escalabilidad.

Análisis del patrón

- **Agilidad:** baja, los cambios normalmente afectan a varias capas y son lentos
- Despliegue: lento, un cambio en un componente puede requerir el despliegue de toda la aplicación.
- Pruebas: el aislamiento entre capas facilita las pruebas
- Rendimiento: generalmente bajo, por el sobrecoste de la comunicación entre capas
- Escalabilidad: baja, si las capas no se distribuyen entre varios nodos
- Desarrollo: fácil, es un patrón muy conocido y extendido. Permite repartir el trabajo de cada capa a distintos expertos (BBDD, diseño de GUI, etc.)

4. Lógica de negocio y acceso a datos

Supuesto inicial

Queremos implementar la funcionalidad “Procesar pedido”

- Un pedido se compone de varias líneas de pedido
- Cada línea de pedido está asociada a un producto e indica cuántas unidades de producto se van a vender
- Si no hay suficientes unidades del producto en stock, el pedido no se puede procesar
- El formulario para crear el pedido permitirá buscar productos por categoría

Patrones de lógica de negocio

Tres opciones:

- Transaction script
- Table module
- Domain model

Transaction script

“Organiza la lógica de negocio en procedimientos, donde cada procedimiento gestiona una petición de la capa de presentación”

- Es la forma más sencilla de organizar la lógica de negocio
- Se crea una transacción para cada una de las funcionalidades de la aplicación
- Cada transaction script se organiza en un único método, haciendo llamadas directamente a la base de datos.

Table Data Gateway

Si queremos evitar hacer llamadas SQL directamente desde los transaction script, podemos combinarlos con el patrón de acceso a datos Table Data Gateway: *“Un objeto que actúa como pasarela para una tabla de la base de datos. Una única instancia gestiona todas las filas de la tabla”*.

Transaction Script + Table Data Gateway

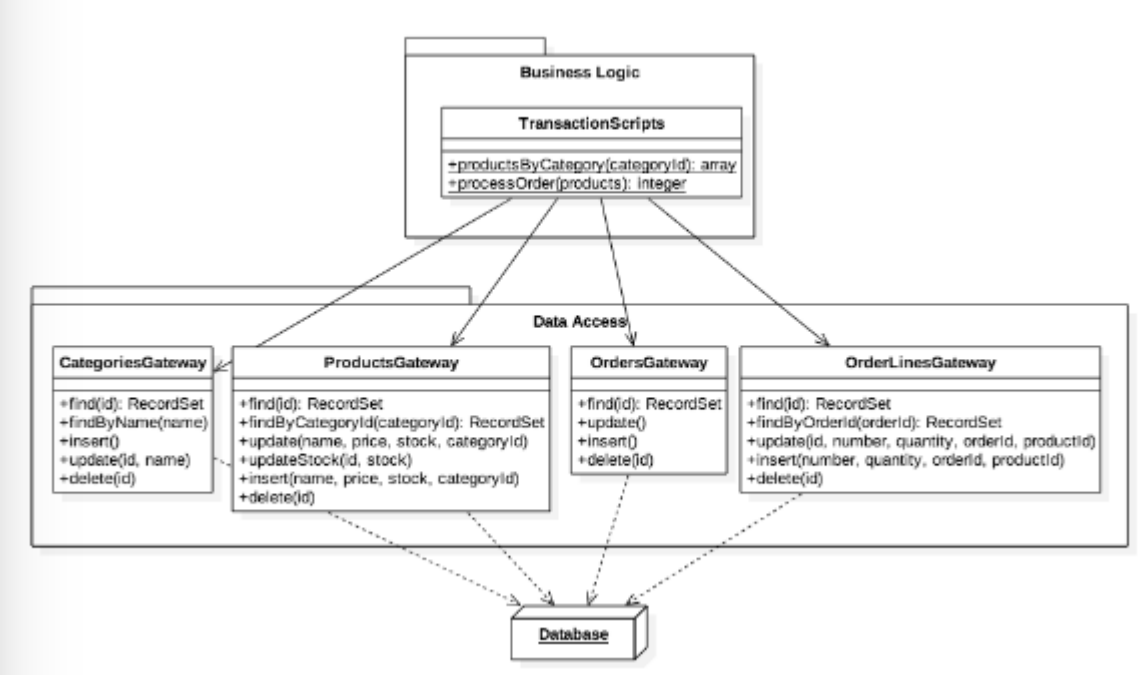


Table module

“Una única instancia gestiona la lógica de negocio para todas las filas en una tabla o vista de la base de datos”

Una única instancia del objeto Table Module permite gestionar todos los registros de la tabla. Todos los métodos excepto insert() necesitan un identificador para localizar el registro en la base de datos. Los objetos table module tambińe pueden usar el patrón Table Date Gateway para comunicarse con la base de datos.

- Table Module implementa la lógica de negocio
- Table Date Gateway implementa el acceso a datos

Table Module

Los objetos Table Module sólo contienen la lógica de negocio que corresponde a cada objeto individual. Las operaciones complejas deben ir en la capa de servicio.

Domain model

Cuando la lógica de negocio es compleja la mejor opción es implementar un modelo de dominio: *“Un objeto del modelo de dominio incorpora tanto el comportamiento como los datos.”*

Transformar el modelo de dominio a una estructura de tablas relacional puede ser complejo.

- Los modelos sencillos tienen un objeto por cada tabla
- Modelos más complejos pueden tener una estructura distinta a la de la base de datos

Dependiendo de la complejidad el acceso a datos se puede hacer de dos maneras.

- Modelos sencillos -> Active Record
- Modelos complejos -> Data Mapper

ActiveRecord
"Un objeto que envuelve una fila de una tabla a vista, encapsula el acceso a la base de datos y añade comportamiento a esos datos."

Data Mapper
Si modelo de dominio es complejo conviene mantenerlo separado del acceso a datos. En estos casos se usa el patrón Data Mapper: *"Una capa de mapeadores se encarga de mover datos entre los objetos y la base de datos, manteniéndolos independientes entre ellos e independientes del mapeador."*

5. Mapeo objeto-relacional

Cuando se decide implementar un modelo de dominio hay que encontrar la manera de adaptar la estructura del modelo a la estructura de la base de datos.

- Las bases de datos orientadas a objetos son una forma natural de persistir objetos, pero no son muy comunes
- Las bases de datos relacionales son mucho más comunes, pero tienen una estructura distinta a los medelos orientados a objetos, y usan SQL para realizar las consultas.

Los sistemas de mapeo objeto-relacional (Object-Relational Mapping, OMR) se encargan de almacenar y recuperar los objetos en bases de datos relacionales. Los sistemas ORM existentes normalmente implementan uno de los dos patrones para persistir modelos de dominio.

- ActiveRecord
    - (+) es más sencillo de implementar
    - (-) incrementa el acoplamiento con la base de datos y dificulta la refactorización
    - (-) es difícil optimizar el SQL y puede perjudicar el rendimiento
- DataMapper
    - (+) permite realizar transformaciones complejas
    - (+) facilita la optimización del acceso a la base de datos
    - (-) es más difícil de entender y configurar

Dos estrategias:

- **Model first:** primero se diseña el modelo de dominio, y luego se crea la base de datos con la estructura necesaria. Situación ideal para usar el patrón ActiveRecord.
- **Database first:** se parte de una base de datos ya existente, por lo que hay que adaptar el modelo de dominio a su estructura. En estos casos puede ser más conveniente usar un Data Mapper.

Patrones de comportamiento

Los patrones de comportamiento gestionan cómo se almacenan y recuperan los objetos de una base de datos relacional. Problemas a resolver:

- Se debe mantener la integridad referencial cuando se crean y modifican múltiples objetos (**Unit of Work**)
- No debe haber múltiples copias del mismo objeto en memoria (**Identity Map**)
- Cargar los objetos relacionados en un modelo de dominio puede acabar recuperando la base de datos completa (**Lazy Load**)

Unit of Work

"Mantiene una lista de objeto afectados por una transacción y coordina la escritura de los cambios y la resolución de problemas de concurrencia".

Llamar a la base de datos para cada cambio en el modelo de dominio puede afectar el rendimiento. En lugar de llamar directamente a la base de datos, el sistema notifica al objeto *Unit of Work,* de manera que pueda llevar un registro de los objetos nuevos, recuperados de la base de datos, modificados y borrados. Cuando es necesario, abre una transacción con la base de datos y realiza todos los cambios en orden.

Identity Map

"Asegura que cada objeto se carga una única vez llevando un registro en un mapa de cada objeto. Busca los objetos en el mapa de recuperarlos de la base de datos."

Cargar un registro varias veces a la base de datos puede dar lugar a múltiples objetos con los mismos datos. Problemas:

- Gasto excesivo de recursos
- Inconsistencia cuando uno de ellos se modifica

El objeto *Identity Map* lleva un registro de los objetos cargados y devuelve referencias para los que ya existe una instancia en memoria. También actúa de caché para la base de datos.

Lazy Load

"Un objeto que no contiene datos, pero sabe cómo obtenerlos."

Cuando se carga un objeto del modelo de dominio puede ser útil cargar sus objetos relacionados. Sin embargo, esto puede degradar el rendimiento si el número de objetos relacionados es elevado, o incluso acabar cargando la base de datos completa en una reacción en cadena. Lazy Load sólo carga un objeto cuando se usa.

Alternativas de implementación:

- **Lazy initialization**: usa un valor nulo para los objetos hasta que se cargan, necesita que las propiedades estén encapsuladas en métodos get
- **Virtual proxy**: los objetos se sustituyen por un objeto vacío que carga los datos cuando es necesario, permite separar la lógica necesaria para cargar los datos de los objetos del modelo de dominio.

Patrones estructurales

A veces no es necesario mapear todos los objetos de dominio a tablas en la BBDD.

- Los objetos pequeños se pueden guardar junto a su contenedor.
- Las colecciones o jerarquías de pequeños objetos se pueden guardar juntas, de manera que se pueda acceder a ellas en una sola operación.

Embedded Value

“Mapea un objeto en varios campos de otra tabla.”

Serialized LOB

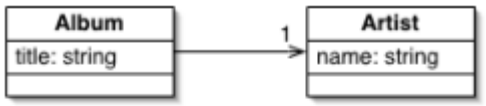
En ocasiones hay objetos que tiene una propiedad que contiene una estructura jerárquica de pequeños objetos. Una forma eficiente de almacenarlos es transformarlos en un único objeto grande (LOB), ya sea en forma binaria (BLOB) o textual (CLOB), y almacenarlo en una única columna.

Los campos BLOB no deberían utilizarse para almacenar imágenes, es preferible almacenar las imágenes en una carpeta y guardar su ruta en la base de datos.

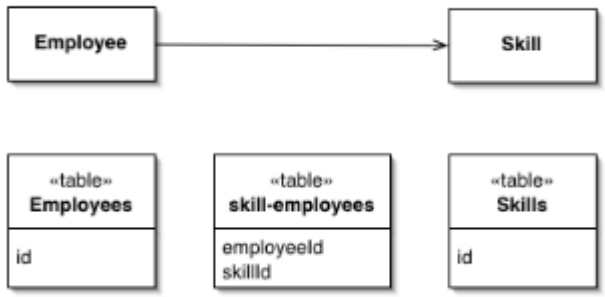
Mapeo de relaciones

Cuando un objeto contiene colecciones o referencias (compartidas) a otros objetos, no deben guardarse como valores en la misma tabla. Necesitamos representar esas referencias para mantener la base de datos en forma normal.

- Mapeo de clave ajena: *“Mapea una asociación entre objetos a una referencia de clave ajena entre tablas.”*



- Mapeo de tabla de asociación: *“Guarda una asociación como una tabla con claves ajenas a las tablas que están vinculadas por la asociación”.*



- Mapeo de la herencia: Las bases de datos relacionales no soportan la herencia. Si decidimos implementar una jerarquía de herencia en el modelo de dominio necesitamos:
    - Podemos usar las clases hijas como si se tratase de la clase padre (polimorfismo)
    - Que todas las clases de la jerarquía compartan un campo identificador común

Patrón Class table inheritance
Se usa una única tabla para cada clase en la jerarquía. Cada tabla almacena únicamente los atributos nuevos

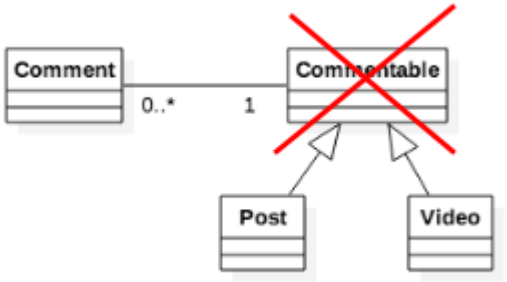
- (-) Necesita hacer join para cada consulta, es un problema para el rendimiento
- (+) No hay columnas irrelevantes

Patrón Concrete table inheritance
Se usa una única tabla para cada clase “hoja”. Cada tabla almacena todos los atributos (heredados + nuevos).

- (-) Es complicado gestionar los identificadores si queremos todas las subclases compartan campo identificador
- (-) No se pueden representar relaciones con las clases abstractas
- (+) No hay columnas irrelevantes
- (+) No necesita hacer join

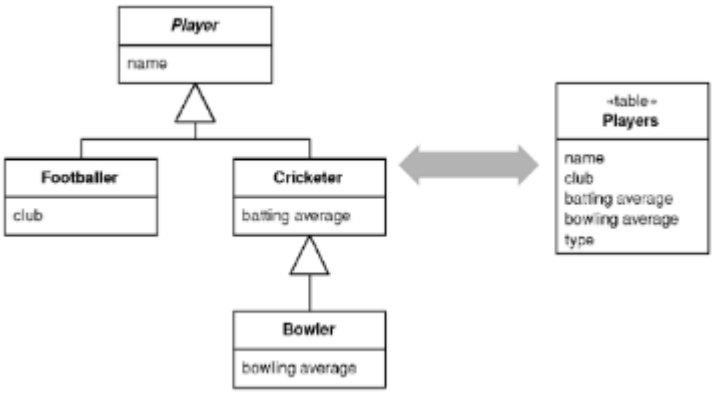
Implementación en Laravel medainte relaciones polimórficas.

- Permite acceder a objetos de distinto tipo a través de una relación ($comment->commentable devolverá un objeto que podrá ser un Post o un Video)
- La clase base no se implementa



Patrón Single table inheritance

- Se usa una única tabla para todas las clases.
- Almacena la unión de todos los atributos de las clases hijas
    - (-) Las columnas que no usan todas las clases gastan espacio
    - (+) Evita joins innecesarios
    - (+) Mismo campo identificador para todas las subclases



Laravel tiene una extensión que permite implementar este patrón manteniendo la jerarquía de clases en el código.

- No permite instanciar objetos de la clase padre -> no podemos usar el polimorfismo $user = User::find(1)

En la práctica es más sencillo juntar todas las clases hijas una sola con la union de todos los atributos y métodos, y un atributo y métodos, y un atributo adicional indicando el tipo de cada objeto.

Autentificación de usuarios

Requisitos:

- En un foro podemos tener usuarios normales y moderadores
- Los usuarios normales se pueden convertir en moderadores a propuesta de otros moderadores

¿Problemas?

- No podemos cambiar el tipo de un usuario, hay que destruirlo y crear uno nuevo
- Tendríamos que hacer comprobación de tipos (instanceof) para comprobar el tipo de usuario cada vez que se ejecuta una funcionalidad

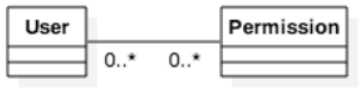
Cuando el usuario pide acceso a una funcionalidad se hacen dos comprobaciones:

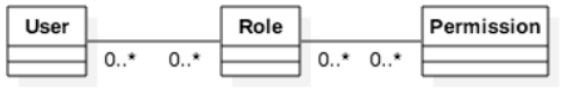
- Autentificación: comprueba la identidad del usuario (recuperando el objeto User de la sesión activa)
- Autorización: comprueba si el usuario identificado tiene permisos para acceder a la funcionalidad solicitada

Al dibujar una pantalla se muestran únicamente los elementos a los que el usuario tiene acceso. Si la autentificación falla se redirige al usuario al formulario de login y cuando el usuario introduce las credenciales correctas se crea una instancia de la clase User y se

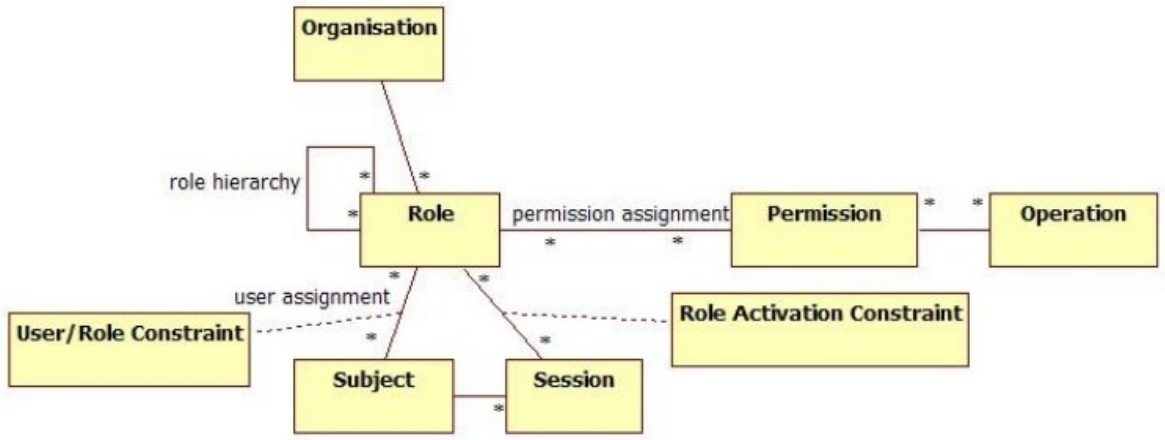
almacena la sesión (objeto global que almacena información compartida entre todas las pantallas)

La clase User no implementa las funcionalidades, se usa para otorgar acceso a las pantallas que las ofertan y se puede simplificar el diseño usando una única clase. Si queremos tener mayor control sobre lo que pueden hacer los usuarios podemos añadir permisos.





Generalización: **patrón Role-Based Access Control (RBAC)** (estándar NIST)



6. Patrones de capa de presentación

Introducción

En una arquitectura en capas, la Capa de Presentación tiene las siguientes responsabilidades:

- Gestionar la interacción con el usuario
- Comunicarse con las capas inferiores que proveen las funcionalidades deseadas
- Mostrar/actualizar la información como resultado de las llamadas a la lógica de negocio

Para estructurar correctamente el código de la capa de presentación conviene distribuir estas responsabilidades entre distintos objetos. Los patrones estructurales más usados para la capa de presentación son: [Fowler, 2003, Osmani, 2015]

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-viewModel (MVVM)

Para todos ellos se considera que el Modelo representa los objetos del modelo de dominio en la capa de lógica de negocio. Los demás objetos (Vista y Controlador/Presenter/ViewModel) pertenecen a la capa de presentación.

Estos patrones son independientes del tipo de aplicación (escritorio, web, móvil, …), dependiendo del framework y de las características de la aplicación será más sencillo emplear unos u otros.

- Normalmente los frameworks para aplicaciones web de servidor están preparados para usar el patrón MVC
- La mayoría de frameworks para aplicaciones web cliente (frontend) permite usar cualquiera de los tres.

Patrón Model-View-Controller

Aunque aparentemente se trata de un patrón sencillo, es uno de los patrones con más variantes y sobre el que menos consenso hay. Hoy en día es el patrón más utilizado en aplicaciones web, aunque de forma muy distinta a como fue diseñado en sus orígenes.

El patrón MVC fue concebido inicialmente para pequeños componentes gráficos en Smalltalk-80. Cada componente gráfico (cuadro de texto, checkbox, etc.) tiene una vista y un controlador:

- La vista se encarga de dibujar el componente (había que escribir el código para esto) a partir de los datos del modelo
- El controlador recibe la interacción del usuario y manipula el modelo

Hay dos posibilidades para implementar la forma en que se actualiza la vista:

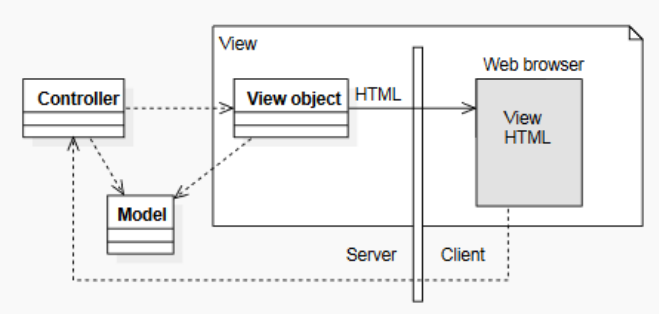
1. El controlador actualiza la vista después de manipular el modelo
2. El modelo notifica a la vista para que se actualice (**modelo activo**)

En aplicaciones web simples (sin clientes enriquecidos con JavaScript) la comunicación entre objetos es distinta:

- La vista gestiona la interacción con el usuario y notifica al controlador
- El controlador manipula el modelo y decide qué vista mostrar a continuación.

Esta forma de comunicarse es necesaria porque la aplicación está dividida físicamente entre el cliente y el servidor. La vista tiene dos partes:

- Un objeto (dinámico) que se instancia en la capa de presentación del servidor para generar el HTML
- El HTML (estático) mostrado en el navegador del cliente

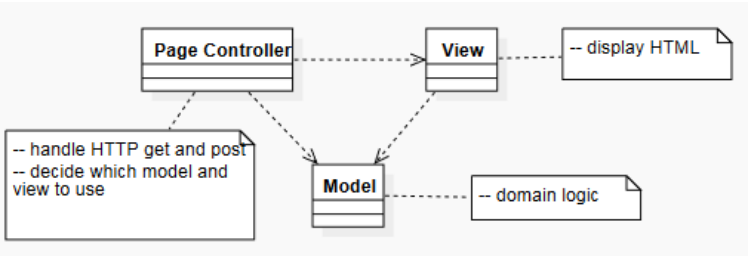


Hay varias formas de programar los controladores:

- Page controller
- Front controller

Un **Front Controller** puede colaborar además con un **Application Controler**.

Patrón Page Controller [Fowler, 2003]: Un objeto que gestiona una petición para una paginación o acción específica en un sitio web.

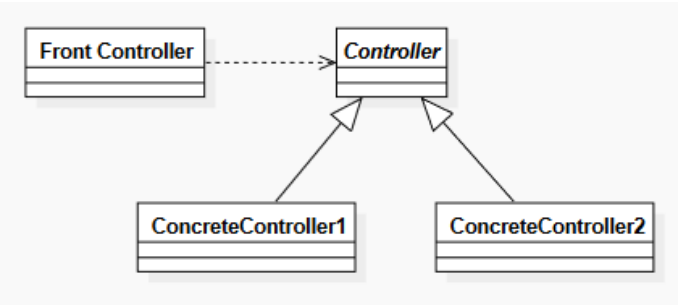


Es el patrón utilizado por ASP.NET (code-behind). Implica la creación de un controlador para cada página lógica de la aplicación. Responsabilidades:

- Decodificar la URL y extraer los datos de la petición
- Crear e invocar los objetos del modelo
- Seleccionar la vista a mostrar

Es manejable cuando la navegación es simple. Para aplicaciones más complejas es muy difícil de gestionar.

Patrón Front Controller [Fowler, 2003] : Un controlador que gestiona todas las peticiones de un sitio web.



El **Front Controller** realiza el comportamiento común a todas las acciones, y luego delega en controladores específicos para cada acción. La principal ventaja frente al patrón Page Controller es que ahora un solo objeto controlador gestiona todas las peticiones para evitar duplicación de comportamiento.

En frameworks web modernos, el Front Controller delega algunas de sus responsabilidades en otros objetos:

- Objetos **Middleware** para chequeos de seguridad, internacionalización, etc.
- Un enrutador (**Router**) para seleccionar el controlador específico que corresponde a cada acción.
- Si la lógica para gestionar la navegación es compleja se puede usar un **Application Controller**.

Cada objeto Middleware se especializa en realizar un tipo de comprobación:

- Verificar si el usuario está autentificado
- Comprobar si el usuario tiene permisos para ejecutar la acción
- Limitar el acceso a los recursos por número de accesos o ancho de banda (*throttling*)
- Comprobar y/o modificar los valores de entrada
- ...

Los frameworks incorporan algunos Middleware por defecto (p. ej. comprobar usuario autentificado). También se pueden crear objetos Middleware personalizados. Se pueden encadenar distintos objetos Middleware para realizar varias comprobaciones antes de ejecutar el controlador:

- Si las comprobaciones son correctas pasa a la ejecución al siguiente Middleware, o finalmente al controlador.
- Si alguna de las condiciones definidas en los Middleware no se cumple se puede detener la ejecución. Por es se les llama también **Filtros**.

El objeto Router es el responsable de seleccionar el controlador apropiado para cada petición. En algunos frameworks como Laravel las rutas se especifican en un archivo de rutas que el Router carga al iniciar la aplicación:

```
Route::get('/','HomeController@index');
Route::get('post/create','PostController@create');
Route::post('post','PostController@store');
```

Otros frameworks como ASP.NET MVC están configurados para hacer una correspondencia entre la ruta y el método del controlador a ejecutar. Por ejemplo, la ruta /Product/Edit/3 se corresponde con:

- Controlador = Product
- Método = Edit
- id = 3

En ocasiones un controlador no puede decidir directamente cuál es la siguiente vista a mostrar, ya que puede depender del estado de los modelos o de la ejecución de una regla de negocio.

Ejemplo
Para una compañía aseguradora, después de modificar los datos de un parte tras la actuación de un especialista podrían pasar dos cosas: ● *Si el incidente se ha resuelto mostraría la vista para cerrar el parte* ● *Si es necesaria la actuación de otro especialista se mostraría la vista correspondiente.*

En estos casos es conveniente mantener un **Application Controller** en una capa separada, p. ej. en la capa de lógica de dominio. Si el flujo de trabajo es complejo podría ser útil usar

una máquina de estados, representada por algún tipo de metadatos.

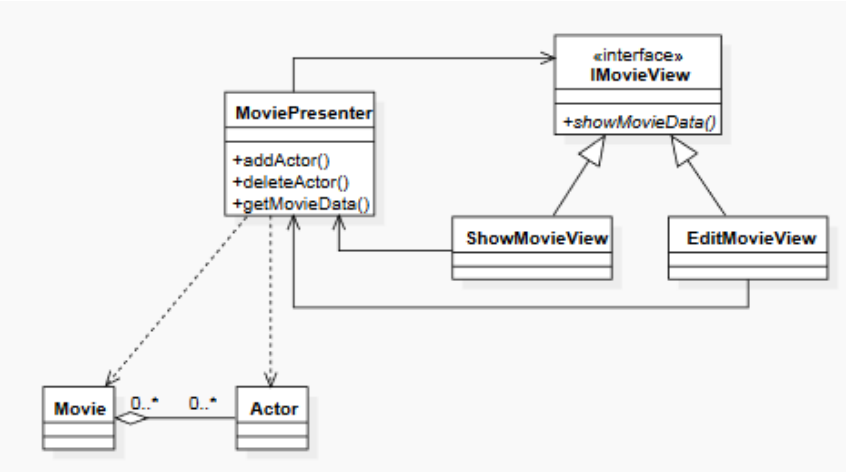
Patrón Model-View-Presenter

El patrón MVC tiene varios inconvenientes:

- La vista está acoplada al modelo, si no tomamos precauciones podemos acabar con vistas que deben hacer un procesamiento complejo para mostrar los datos.
- Cada vista tiene su propio controlador, hay poca reutilización de código
- Es difícil probar los controladores porque están acoplados con vistas concretas

Para solucionar estos problemas, el patrón Model-View-Presenter (MVP) propone el uso de un objeto **Presenter** en lugar del controlador de MVC.

La vista ya no depende del modelo, ahora el objeto Presenter lee el modelo y prepara los datos para pasárselos a la vista. Para facilitar la reutilización de código, los objetos Presenter están desacoplados de las vistas concretas que lo usan. De esta manera también es más fácil probar los Presenter **inyectándoles** vistas mock.



Cuando se crea una instancia de la vista, ésta crea a su vez un Presenter para que cargue la información que necesita. Si la aplicación es distribuida el Presenter realiza la petición en un hilo paralelo, de manera que la vista puede continuar su ejecución, sin bloquear la interfaz. Cuando el presenter recibe la información, pasa los datos a la vista para que los muestre. Cada vez que una vista necesita nueva información se la pide al Presenter siguiendo el mismo procedimiento.

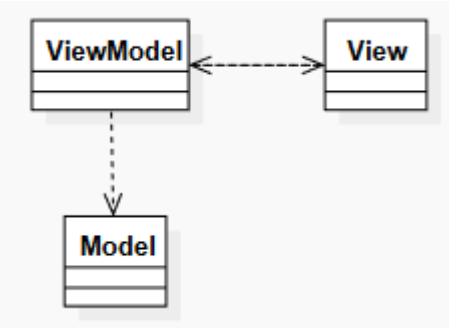
Patrón Model-View-ViewModel

Cuando una aplicación tiene mucha interacción en el interfaz, la complejidad del código puede aumentar demasiado:

- Una sola vista puede necesitar datos de muchos modelos
- Hay que gestionar todos los eventos de los controles gráficos del interfaz y responder adecuadamente a cada acción
- Un cambio en el modelo puede necesitar que se actualicen varias partes de una

pantalla

Para este tipo de aplicaciones surgió el patrón Model-View-ViewModel (MVVM).



El objeto **ViewModel** es una representación del modelo en la capa de presentación, que además contiene la lógica necesaria para responder a los eventos del interfaz.

El objeto ViewModel es el responsable de comunicarse con la capa de lógica de negocio cuando se solicita desde la vista. El verdadero potencial de este patrón está en el uso de **Data Binding**, creando un enlace entre los controles del interfaz y los objetos Viewmodel:

- Cuando el usuario actualiza el campo de un formulario, se actualiza automáticamente la propiedad asociada del View Model.
- Cuando el ViewModel se actualiza como resultado de una llamada a la lógica de negocio, se actualizan automáticamente los controles asociados en el interfaz.

Esto permite construir interfaces con mucho menos código.

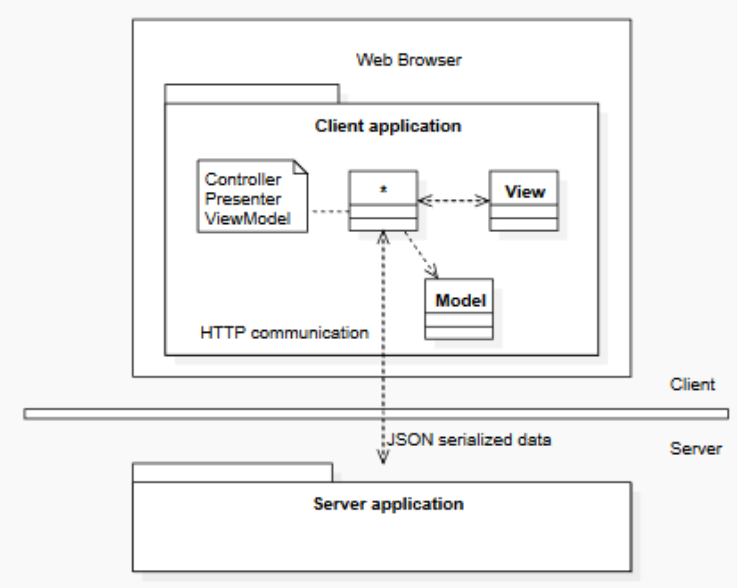
Aplicaciones web RIA

Rich Internet Application (RIA): Una Aplicación Rica (o enriquecida) de Internet es una aplicación web que tiene un interfaz con características similares a las aplicaciones de escritorio.
Se pueden usar distintas tecnologías para implementarlas: JavaScript, Java Applets, Flash, Silverlight, … Las más extendidas actualmente son las aplicaciones de tipo **Single Page Application (SPA)**, y usan JavaScript y AJAX en el cliente.

Funcionamiento de una aplicación SPA:

- El cliente carga un HTML mínimo y los scripts con la lógica de la aplicación cliente
- El interfaz se construye dinámicamente, generando el HTML necesario a partir de los datos proporcionados por el servidor.
- El cliente realiza peticiones AJAX al servidor, la página no se recarga
- Los datos se transmiten serializados en formato JSON

Independientemente de la arquitectura usada en el servidor, estas **aplicaciones cliente necesitan estructurar su código**. Para eso existen frameworks que favorecen el uso de los distintos **patrones MV***.



7. Diseño web adaptable

Introducción

- **Adaptive web design**: Técnicas para adaptar el contenido de una página web a algún dispositivo
- **Responsive Web Design (RWD)**: Técnicas para que el contenido de una página web se adapte automáticamente el tamaño de pantalla del dispositivo

En español normalmente se usa “Adaptable” como sinónimo de “Responsive”. Es necesario para mejor experiencia del usuario (UX) -> disminuye la tasa de rebote

“Por primera vez en España hay más usuarios de internet (76,2%) que de ordenador (73,3%). El 77,1% de los internautas accedieron a internet mediante el teléfono móvil.

Conceptos

Viewport
Muchas páginas web están diseñadas para escritorio, con un tamaño fijo en píxeles. Para poder mostrarlas correctamente, los navegadores móviles dibujan la página en un área (**viewport**) de tamaño similar al de una pantalla de escritorio: unos 980px, aunque puede variar.
El tamaño del viewport se puede controlar con la etiqueta <meta>

<meta name="viewport"content="width=device-width, initial-scale=1">

Densidad de pantalla
No todos los dispositivos tiene la misma resolución con el mismo tamaño de pantalla.

- Densidad de pantalla: Cantidad de píxeles que caben en la pantalla.

Normalmente se miden en puntos por pulgada (dots per inch, dpi).

- ldpi (120dpi)
- mdpi (160dpi)
- hdpi (240dpi)
- xhdpi (320dpi), xxhdpi (480dpi), xxxhdpi (640dpi), ...

Usando píxeles reales, una página con diseño fijo se vería muy pequeña en dispositivos con densidades de pantalla altas. Para que las páginas sean legibles, los navegadores móviles aumentan el tamaño de los píxeles CSS. Se usa como referencia el tamaño real de un píxel en dispositivos con densidad mdpi (160dpi).

Media queries

- **Media queries**: introducidas en CSS3, permiten aplicar un conjunto de reglas sólo si se cumplen unas determinadas condiciones. Permiten identificar el tipo de dispositivo, el tamaño y la orientación de la pantalla.

```
@media only screen and (max-width: 500px) {
    body {
        background-color: lightblue;
    }
}
```

Sistemas de rejillas

Muchas páginas estructuran el contenido mediante una rejilla (grid), que divide la página en columnas. El uso de rejillas facilita el posicionamiento de los elementos en la pantalla. Normalmente se definen rejillas de 12 columnas, que se redimensionan automáticamente al cambiar el tamaño de la pantalla.

Fluid grids: Una “rejilla fluida” reestructura el contenido apilando las columnas cuando el tamaño de la pantalla es demasiado pequeño.

HTML semántico

Etiquetas semánticas en HTML5:

- Header: Grupo de ayudas introductorias o de navegación
- Nav: Menú de navegación
- Main: Contenido principal
- Article: Composición auto-contenida en un documento
- Aside: Información adicional relacionada con el contenido principal
- Footer: Pie de página

Su uso ayuda a posicionar de forma más inteligente el contenido de la página web.

Imágenes responsive

- Técnica básica: max-width
- Distintas imágenes para distintas densidades de pantalla
- Adaptar el nivel de detalle al tipo de pantalla

Rendimiento de la página

Servir el mismo contenido independientemente del dispositivo puede afectar negativamente al tiempo de carga en conexiones móviles. El contenido que no se muestra en pantallas

pequeñas o las imágenes con resolución alta incrementan innecesariamente el tiempo de carga.

Estrategias de diseño

Planificación

- **No diseñar para un ancho de pantalla específico**, las diferencias de tamaño de pantalla y densidad entre distintos dispositivos puede hacer que el resultado no se vea bien.
- **Identificar qué contenidos son importantes y cuales son** prescindibles para ocultarlos en las versiones móviles
- **Realizar mockups para las distintas versiones**
- **Probar el resultado en dispositivos reales**
- **Evitar plugins siempre que sea posible**

Mockups

Diseña primero para móviles

- Mejora el rendimiento en dispositivos pequeños
- En lugar de cambiar los estilos cuando la pantalla es más pequeña, lo hace cuando es más grande
- Intenta cargar el contenido que se ve a simple vista (adove-the-fold) en 1 segundo:
    - Situar primero el HTML y CSS imprescindibles (no más de 14KB) para conexiones 3G.
    - A continuación, cargar JavaScript e imágenes secundarios.
- Usar minificadores de código y optimizadores de imágenes
- Usar carga condicional de contenido e imágenes:
    - Carga condicional con JavaScript
    - Devolver imágenes de distinto tamaño desde el servidor.

Herramientas

Frameworks

Paquete de ficheros CSS y JavaScript con clases CSS predefinidas para formatear el contenido. Se adaptan automáticamente al tamaño de pantalla. Frameworks CSS más utilizados:

- Bootstrap
- Foundation

Plantillas

Páginas prediseñadas para modificar únicamente el contenido

Herramientas de desarrollo

Herramientas de desarrollo incluidas en los navegadores:

- Firefox Developer Tools / Firefox Developer Edition
- Chrome DevTools

Permiten inspeccionar el contenido de una página, las reglas CSS que se aplican a cada elemento y ver la página en distintos tamaños de pantalla.

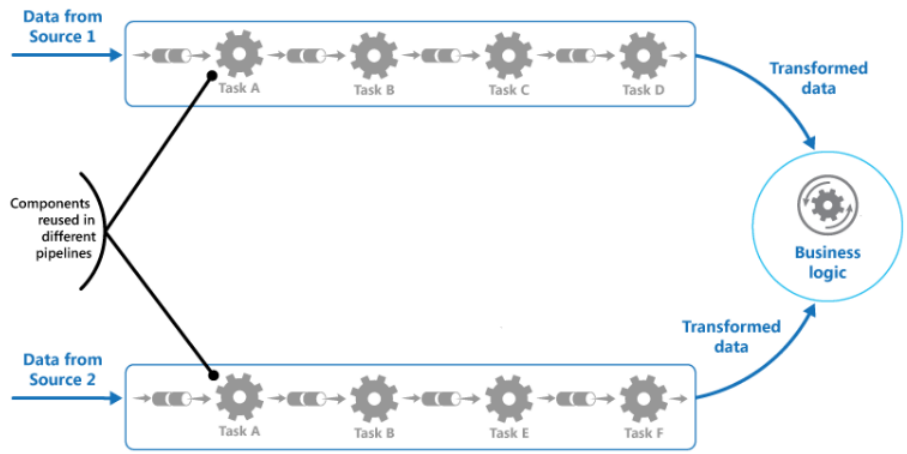
Optimización para móviles

- Prueba de optimización para móviles de Google
- Yslow de Yahoo

8. Otros patrones arquitecturales

Arquitectura tuberías y filtros

Los datos pasan por una serie de filtros, que transforman la información.



Muy usado en sistema Unix, aunque no tiene por qué implementarse necesariamente mediante tuberías y línea de comandos. Filosofía Unix:

- Modularidad
- Simplicidad
- Composición mediante el encadenamiento de programas sencillos para realizar tareas complejas
- “Haz una cosa y hazla bien”

Ejemplo: Apertium

Beneficios:

- Facilita el diagnóstico
- Permite añadir fácilmente nuevos filtros entre dos módulos
- Desarrollo de aplicaciones derivadas mediante la reutilización de módulos (interNOSTRUM -> Apertium)

Análisis del patrón

- **Agilidad**: alta, debido a la modularidad
- **Despliegue**: lento, normalmente se trata de aplicaciones que se ejecutan en las máquinas de los usuarios finales
- **Pruebas**: fácil, se pueden probar los filtros de forma independiente

- **Rendimiento**: alto, aunque depende de la complejidad de los filtros
- **Escalabilidad**: baja, distribuir los filtros en distintos nodos añade complejidad
- **Desarrollo**: fácil, si se usan los mecanismos de comunicación del sistema operativo (tuberías)

Arquitectura microkernel

También conocida como arquitectura de *plugins,* consiste en un núcleo central que se puede extender mediante módulos

Componentes

- Núcleo central: contiene la funcionalidad mínima para que el sistema funcione
- Plugins
    - Componentes independientes que añaden funcionalidades al núcleo central
    - Puede haber dependencias entre plugins
    - Se suelen organizar en un registro o repositorio para que el núcleo pueda obtener los plugins necesarios

Ejemplo: Eclipse IDE, Sonic Visualizer

Arquitectura microkernel

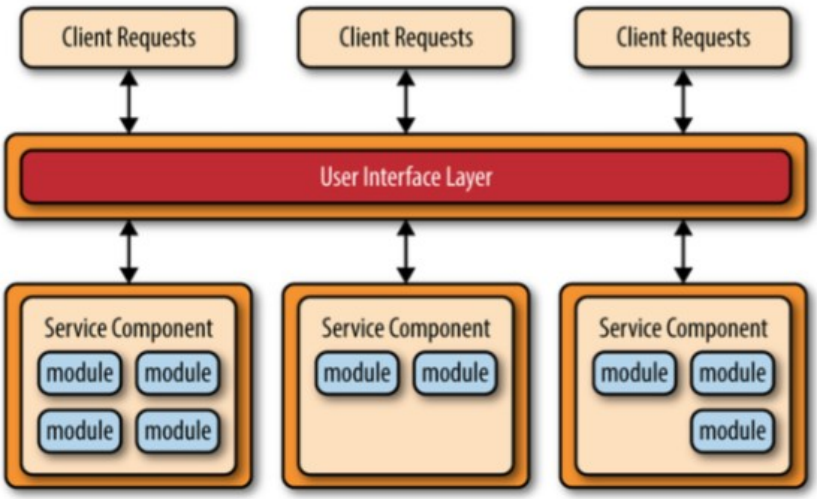
Muy adecuado para el diseño y desarrollo evolutivo e incremental, y aplicaciones “producto”.

Análisis del patrón

- **Agilidad**: alta, es sencillo añadir nuevos componentes
- **Despliegue**: sencillo, se pueden añadir nuevos plugins en tiempo de ejecución
- **Pruebas**: sencillo, se pueden probar los plugins por separado
- **Rendimiento**: normalmente alto, ya que se pueden instalar únicamente los plugins necesarios.
- **Escalabilidad**: baja, normalmente diseñado como un único ejecutable
- **Desarrollo**: difícil, el diseño del interfaz de los plugins debe plantearse cuidadosamente. La gestión de versiones y repositorio de plugins añade complejidad.

Arquitectura de microservicios

Arquitectura distribuida, el cliente se comunica con los servicios mediante algún protocolo de acceso remoto (REST, SOAP, RMI, etc)



La implementación más frecuente utiliza HTTP como protocolo de comunicación, definiendo interfaces REST (REpresentantional State Transfer) para acceder a los servicios.
Ejemplo: Netflix OSS

Además para el desarrollo de aplicaciones y servicios web.

Análisis del patrón

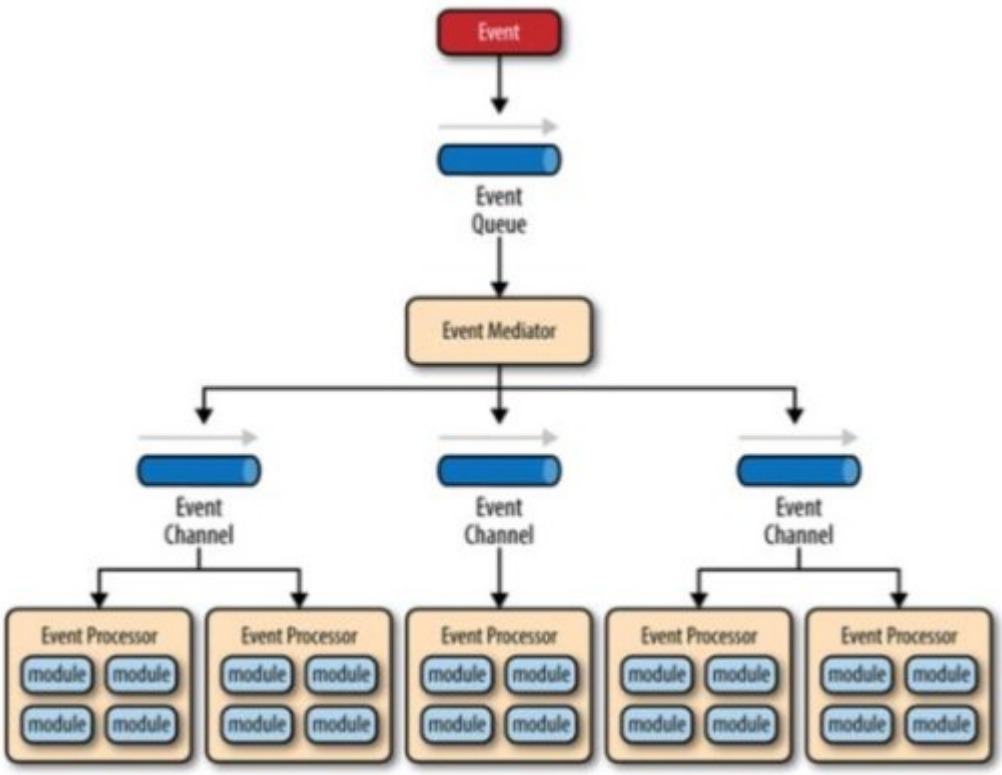
- **Agilidad**: alta, los cambios afectan a componentes aislados
- **Despliegue**: sencillo, favorece la integración continua
- **Pruebas**: sencillo, debido a la independencia de los servicios
- **Rendimiento**: bajo, debido a la naturaleza distribuida
- **Escalabilidad**: alta, permite escalar los servicios por separado
- **Desarrollo**: fácil, la independencia de los servicios reduce la necesidad de coordinación. El uso de protocolos de comunicación estándar facilita el desarrollo

Arquitectura orientada a eventos

El sistema se compone de pequeños componentes que responden a eventos, y de algún mecanismo para gestionar las colas de eventos que se reciben. Dos topologías alternativas:

- Mediador
- Broker

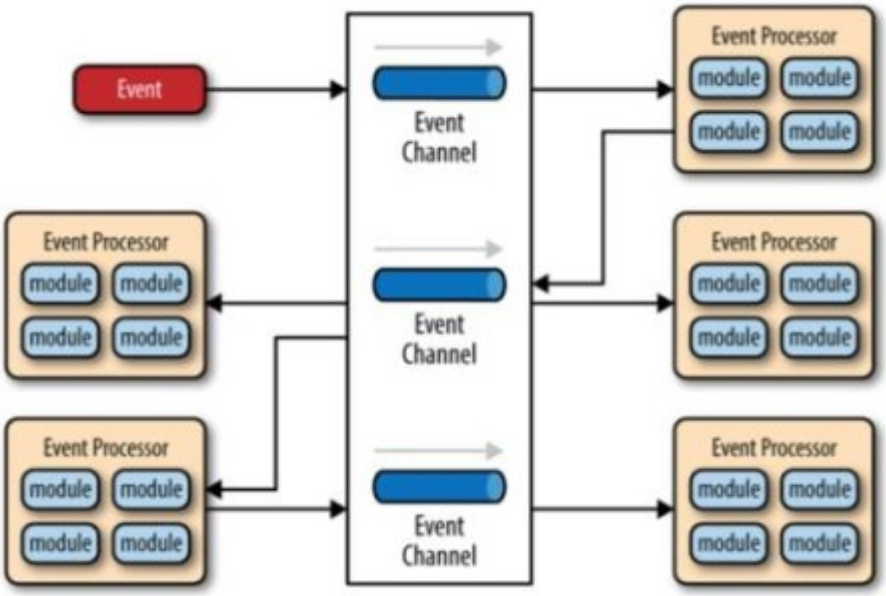
Topología mediador: el procesamiento de eventos implica varios pasos que deben ejecutarse de manera orquestada.



Componentes

- **Procesador de eventos**
  - Contiene la lógica de negocio
  - Pequeñas unidades autocontenidas y altamente independientes del resto
- **Mediador**:
  - Recoge los eventos de inicio y los envía a los procesadores en el orden apropiado según el tipo de evento
  - Se puede implementar mediante soluciones open source, y definir usando lenguajes de definición de procesos (business process execution language, BPEL)

Topología broker: Los procesadores de eventos se encadenan unos con otros mediante eventos que pasan a través del broker. Cuando un procesador de eventos termina su trabajo, genera un evento para que se ejecuten los siguientes componentes.



Componentes

- **Procesador de eventos**
  - Contiene la lógica de negocio
  - Pequeñas unidades autocontenidas y altamente independientes del resto
- **Broker**: más ligero que el mediador, se encarga únicamente de gestionar las colas de eventos para que los procesadores no tengan que preocuparse de los detalles de implementación

Es una arquitectura compleja, de naturaleza asíncrona y distribuida.

Análisis del patrón

- **Agilidad**: alta, los cambios normalmente afectan a uno o pocos componentes
- **Despliegue**: sencillo, debido al bajo acoplamiento. Más complicado en el caso del mediador, ya que se debe actualizar cada vez que hay un cambio en los procesadores de eventos
- **Pruebas**: complicado, debido a la naturaleza asíncrona y la necesidad de herramientas especializadas para generar eventos
- **Rendimiento**: alto, debido a la posibilidad de paralelizar la ejecución de componentes
- **Escalabilidad**: alta, los componentes pueden escalar por separado
- **Desarrollo**: difícil, la gestión de errores es compleja

9. Diseño detallado y patrones GRASP

Perspectiva de análisis vs. Perspectiva de diseño

UML incluye los diagramas de clases para ilustrar clases, interfaces y sus asociaciones. Éstos se utilizan para el modelado estático de objetos. Los diagramas de clases se pueden

usar desde dos perspectivas:

- Perspectiva conceptual -> MODELO DE DOMINIO
- Perspectiva de implementación -> modelado de la CAPA DE DOMINIO

Los diagramas de diseño de clases se derivan del modelo de dominio, añadiendo los métodos y secuencias de mensaje necesarios para satisfacer los requisitos. Por lo tanto tenemos que:

- Decidir qué operaciones hay que asignar a qué clases
- Cómo los objetos deberían interactuar para dar respuesta a los casos de uso

El artefacto más importante del flujo de trabajo de diseño es el Modelo de Diseño, que incluye el diagrama de clases software (no conceptuales) y diagramas de interacción.

Los diagramas de diseño de clases normalmente contienen nuevas clases que no están presentes en el modelo de dominio.

- Clases de utilidad: clases reutilizables
- Librerías: clases del sistema
- Interfaces: definiendo comportamientos abstractos
- Clases de ayuda: aparecen por la descomposición de clases grandes

UML define una responsabilidad como “un contrato u obligación de una clase”. Las responsabilidades se asignan a las clases durante el diseño de objetos:

- Hacer:
    - Hacer algo él mismo (p.ej. crear un objeto, realizar un cálculo)
    - Iniciar la acción en otros objetos
    - Controlar y coordinar actividades en otros objetos.

- Saber o Conocer:
    - Conocer sus datos privados (encapsulados)
    - Conocer los objetos con los que se relaciona
    - Conocer las cosas que puede derivar o calcular

Patrones GRASP

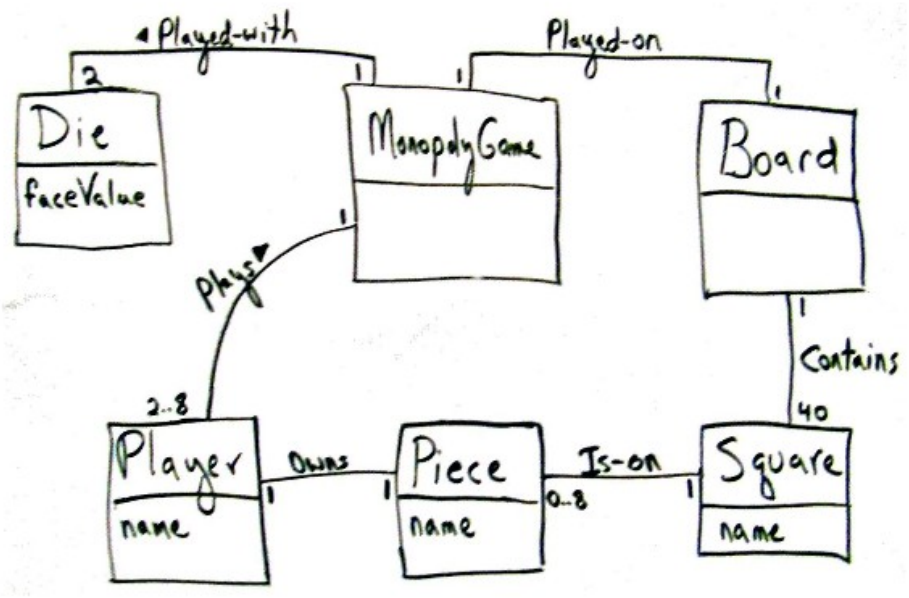
GRASP es un acrónimo para General Responsibility Assignment Software Patterns. Describen 9 principios fundamentales del diseño de objetos y de la asignación de responsabilidades, expresado en términos de patrones. Se dividen en dos grupos:

BÁSICOS	AVANZADOS
● Creador ● Experto (en Información) ● Bajo Acoplamiento ● Controlador ● Alta Cohesión	● Polimorfismo ● Fabricación Pura ● Indirección ● Protección de Variaciones

Para ilustrar los patrones vamos a suponer que queremos modelar un monopoly. Dibujar su modelo de dominio (perspectiva conceptual.

- Nota: asume dos dados

- Nota: comienza modelando el tablero, las casillas, y el acto de hacer una tirada y mover las piezas por parte de un jugador



Protección de variaciones

Problema:

- ¿Cómo diseñar objetos, subsistemas y sistemas de tal manera que las variaciones o inestabilidad en estos elementos no tenga un impacto indeseable sobre otros elementos?

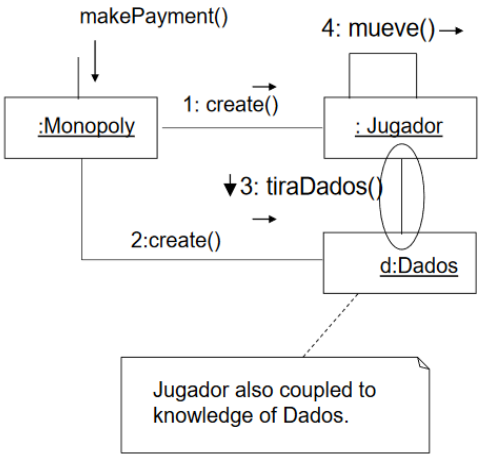
Solución:

- Identifique los puntos de variación o inestabilidad; asigne responsabilidades para crear una interfaz estable a su alrededor.
- Nota: el término interfaz se usa en su sentido más amplio, refiriéndose a los métodos que expone una clase; no implica el uso de interfaces de lenguajes de programación.

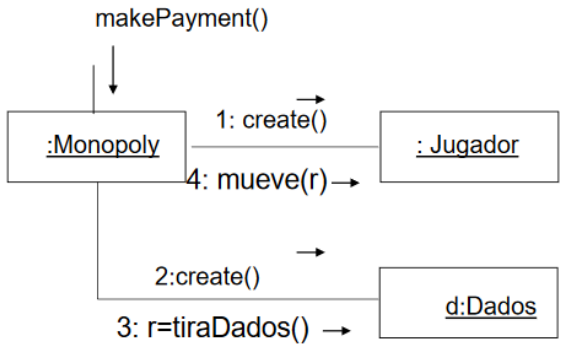
Bajo acoplamiento

Asume que necesitamos modelar el acto de tirar los dos dados de un jugador en el monopoly. ¿A quíen asignamos la responsabilidad?
Parece que Player es el mejor candidato pero, ¿qué problema tiene esta solución?

- Si el jugador tira los dados, es necesario acoplar al Jugador con conocimiento sobre los Dados (acoplamiento que antes no existía)



Una solución alternativa es que sea el propio juego el que tire los dados y envíe el valor que ha salido al Jugador. Se disminuye el acoplamiento entre Jugador y Dados y, desde el punto de vista del "acoplamiento" es mejor diseño.



¿Cómo soportar una baja dependencia, un bajo impacto de cambios en el sistema y un mayor reuso?

- Asigna una responsabilidad de manera que el acoplamiento permanezca bajo.

Acoplamiento: medida que indica cómo de fuertemente un elemento está conectado a, tiene conocimiento, o depende de otros elementos. Una clase con alto acoplamiento depende de muchas otras clases (librerías, herramientas, etc.)

Problemas de alto acoplamiento:

- Cambios en clases relacionadas fuerzan cambios en clases afectadas por el alto acoplamiento.
- La clase afectada por el alto acoplamiento es más difícil de entender por sí sola: necesita entender otras clases.
- La clase afectada por el alto acoplamiento es más difícil de reutilizar, porque requiere la presencia adicional de las clases de las que depende.

Beneficios:

- Las clases con bajo acoplamiento normalmente…

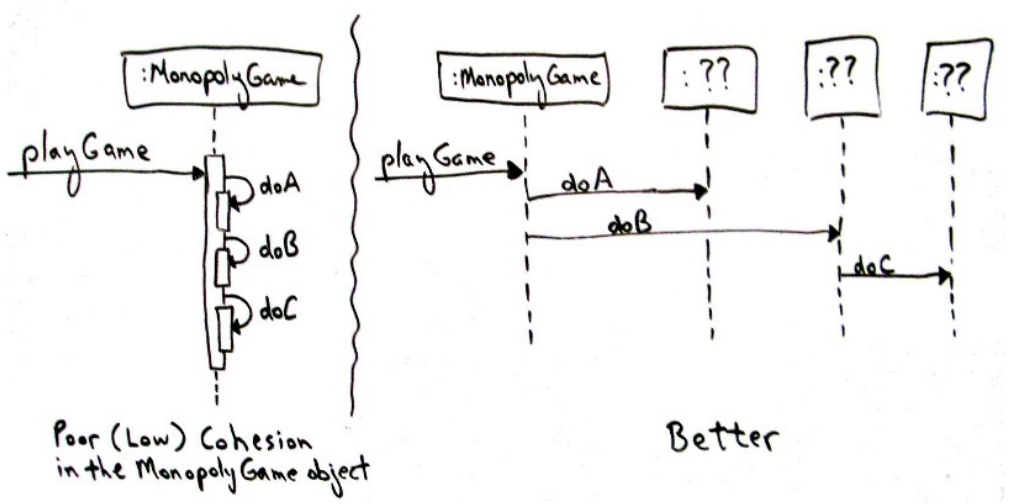
    - No les afectan los cambios en otros componentes
    - Son sencillas de comprender de forma aislada
    - Es fácil reutilizarlas
- Contraindicaciones:
    - El alto acoplamiento con elementos estables es raramente un problema, ya que raramente habrá cambios que puedan afectar a estas clases, por lo que no merece la pena el esfuerzo de evitar este acoplamiento.

Tipos de acoplamiento: El acoplamiento dentro de los diagramas de clases puede ocurrir por varios motivos:

- Definición de Atributos: X tiene un atributo que se refiere a una instancia Y
- Definición de Interfaces de Métodos: p.ej. un parámetro o una variable local de tipo Y se encuentra en un método de X
- Definición de Subclases: X es una subclase de Y
- Definición de Tipos: X implementa la interfaz Y

Alta cohesión

¿Cómo reescribirías el siguiente código del monopoly?



Cohesión: medida de cómo de fuertemente se relacionan y focalizan las responsabilidades de un elemento. Los elementos pueden ser clases, subsistemas, etc. Una clase con baja cohesión hace muchas actividades poco relacionadas o realiza demasiado trabajo. Problemas causados por un diseño con baja cohesión:

- Difíciles de entender
- Difíciles de usar
- Difíciles de mantener
- Delicados: fácilmente afectables por el cambio

¿Cómo mantener la complejidad manejable?

- Asigna las responsabilidades de manera que la cohesión permanezca alta. Clases con baja cohesión a menudo representan abstracciones demasiado elevados, o han asumido responsabilidades que deberían haber sido asignadas a otros objetos.

Beneficios

- Aumenta la claridad y comprensibilidad del diseño
- Se simplifican el mantenimiento y las mejoras

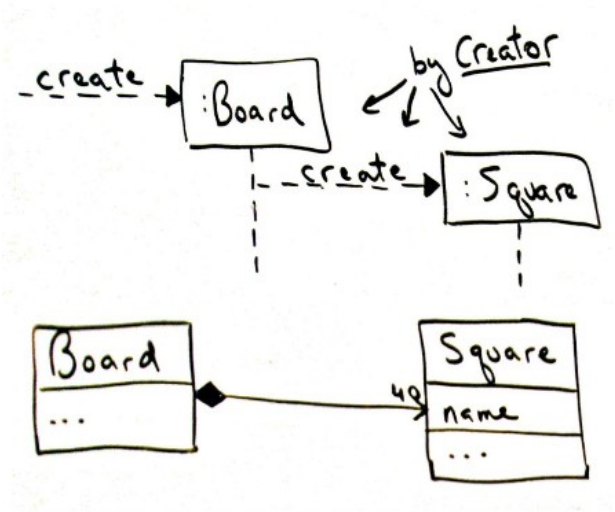
Contradicciones: en ocasiones se puede aceptar una menor cohesión:

- Agrupar responsabilidades o código en una única clase o componente puede simplificar el mantenimiento por una persona (p.ej. Un experto en base de datos)
- Objetos distribuidos entre servidores. Debido al sobrecoste y las implicaciones en el rendimiento, a veces es deseable crear objetos menos cohesivos que provean un interfaz para numerosas operaciones.

Una mala cohesión normalmente implica un mal acoplamiento, y viceversa. Una clase con demasiadas responsabilidades (baja cohesión) normalmente se relaciona con demasiadas clases (alto acoplamiento).

Creador

En el Monopoly, ¿quién debería ser el responsable de crear Casillas?
Puesto que un Tablero contiene casillas, por el patrón Creator éste debería ser el que las creara. Además, una casilla sólo está en un tablero, por lo que la relación es de composición.



¿Quién debería ser el responsable de crear una nueva instancia de alguna clase?

- Asigna a la clase B la responsabilidad de crear una instancia de la clase A si una o más de las siguientes afirmaciones es cierta:
    - B agrega objetos de tipo A
    - B contiene objetos de tipo A
    - B graba objetos de tipo A
    - B tiene datos inicializadores que serán pasados a A cuando sea necesario crear un objeto de tipo A (por tanto B es un Experto con respecto a la creación de A).

Beneficios

- Promueve el bajo acoplamiento, lo que implica menos dependencias (mejor mantenimiento) y mayores oportunidades de reutilización.

Contraindicaciones

- A veces la creación de un objeto requiere cierta complejidad, como por ejemplo el uso de instancias recicladas para aumentar el rendimiento o la creación condicional de una instancia perteneciente a una familia de clases. En estos casos es preferible delegar la creación a una clase auxiliar, mediante el uso de los patrones Concrete Factory o Abstract Factory.

Experto en información

Necesitamos ser capaces de referenciar una casilla particular, dado su nombre (la calle que representa). ¿A quíen le asignamos la responsabilidad?

El patrón Information Expert nos indica que debemos asignar esa responsabilidad al objeto que conoce la información necesaria: los nombre de todas las casillas. Éste objeto es el objeto TABLERO.
¿Cuál es el principio general de asignación de responsabilidades a objetos?

- Asigna cada responsabilidad al experto de información: la clase que tiene (la mayor parte de) la información necesaria para cubrir la responsabilidad. A veces hay que aplicar el Information Expert en cascada.

Beneficios

- Se representa la encapsulación de información, ya que los objetos usan su propia información para completar las tareas. Esto implica normalmente un bajo acoplamiento.
- El comportamiento se distribuye entre las clases que contienen la información necesaria, consiguiendo clases más ligeras.

Contraindicaciones

- ¿Quién debería ser el responsable de almacenar una apuesta en la base de datos? Añadir esta responsabilidad a la clase Apuesta aumentaría sus responsabilidades añadiendo lógica de acceso a datos, disminuyendo así su cohesión.

Controlador

En el monopoly puede haber múltiples operaciones del sistema, que en un principio se podría asignar a una clase System. Sin embargo esto no significa que finalmente deba existir una clase software System que satisfaga este requisito; es mejor asignar las responsabilidades a uno o más Controllers.

Existen dos posibilidades:

- MonopolyGame
- ProcessInitializaHandler (esta solución requiere que haya otros controladores como ProcessPlayGameHandler, etc.)

¿Quién debería ser el responsable de manejar un evento de entrada al sistema?

- ¿Quién es el primer objeto de la capa de dominio que recibe los mensajes de la interfaz?

Asigna la responsabilidad de recibir o manejar un evento del sistema a una clase que represente una de estas dos opciones:

- El sistema completo (Control 'fachada')
- Un escenario de un Caso de Uso (estandariza nomenclatura: ControladorRealizarCompra, CoordinadorRealizarCompra, SesionRealizarCompra, ControladorSesionRealizarCompra)

El Controlador del que habla este patrón NO es el controlador del patrón MVC. Normalmente ventanas, applets, etc. reciben eventos mediante sus propios controladores de interfaz, y los DELEGAN al tipo de controlador del que hablamos aquí.

Polimorfismo

Vamos ahora a incluir el concepto de tipos de casillas en el Monopoly. Distintos tipos de casillas. En función del tipo de casilla, el comportamiento del método caerEn() varía:

- Casilla de suerte: coger una carta de la suerte
- Casilla de propiedades: comprar o pagar
- Casilla de Vaya a la Cárcel: ir a la cárcel
- Casilla de Tasas: pagar tasas
- ...

¿A quién asigno la responsabilidad caerEn()?¿Cómo lo implemento?

¿Cómo manejar alternativas basadas en un tipo sin usar sentencias condicionales if-then o switch que requerirían modificación en el código?

- Cuando alternativas o comportamientos relacionados varían por el tipo (clase), asigna la responsabilidad del comportamiento usando "operaciones polimórficas" a los tipos para los cuales el comportamiento varía.
- Corolaria: No preguntes por el tipo de objeto usando lógica condicional.

Beneficios

- Es fácil extender el sistema con nuevas variaciones
- Se pueden introducir nuevas implementaciones sin afectar a las clases cliente

Contradicciones

- No es raro dedicar demasiado tiempo a la realización de diseños preparados para cambios poco probables, mediante el uso de herencia y polimorfismo.

Fabricación pura

Es necesario guardar instancias del Monopoly en una base de datos relacional. ¿Quién debería tener esa responsabilidad? Por Expert la clase Monopoly debería tener esta responsabilidad, sin embargo:

- La tarea requiere un número importante de operaciones de base de datos, ninguna relacionada con el concepto Monopoly, por lo que Monopoly resultaría incohesiva.
- Monopoly quedaría acoplado con la interfaz de la base de datos (ej. JDBC en Java, ODBC en Microsoft) por lo que el acoplamiento aument.

- Guardar objetos en una base de datos relacional es una tarea muy general para la cual se requiere que múltiples clases le den soporte. Colocar éstas en Monopoly sugiere pobre reuso o gran cantidad de duplicación en otras clases que hacen lo mismo.

Solución:

- Crear una clase (PersistentStorage) que sea responsable de guardar objetos en algún tipo de almacenamiento persistente (tal como una base de datos relacional).

Problemas resueltos:

- La clase Monopoly continua bien definida, con alta cohesión y bajo acoplamiento.
- La clase PersistentStorage es, en sí mísma, relativamente cohesiva, tiene un único propósito de almacenar o insertar objetos en un medio de almacenamiento persistente.
- La clase PersistentStorage es un objeto genérico y reusable.

¿Qué objeto debería tener la responsabilidad cuando no se desean violar los principios de “Alta Cohesión” y “Bajo Acoplamiento” o algún otro objetivo, pero las soluciones que sugiere Experto en información no son apropiadas?

- Asigne un conjunto “altamente cohesivo” de responsabilidades a una clase artificial conveniente que no represente un concepto del dominio del problema, algo producto de la “imaginación” para soportar “alta cohesión”, “bajo acoplamiento” y reuso.

En sentido amplio, los objetos pueden dividirse en dos grupos:

- Aquellos diseñados por/mediante descomposición representacional. (Ej. - Monopoly representa el concepto “partida”)
- Aquellos diseñados por/mediante descomposición conductual. Este es el caso más común para objetos Fabricación pura.

Indirección

¿Cómo podemos desacoplar el juego del hecho de que se juega con 2 dados?

- Cubilete: El objeto cubilete es un ejemplo de indirección: un elemento que no existía en el juego real pero que introducimos para aislarnos del número de dados que usa el juego.

¿Dónde asignar una responsabilidad para evitar acoplamiento directo entre dos o más cosas? ¿Cómo desacoplar objetos de tal manera que el bajo acoplamiento se soporte y el reuso potencial se mantenga alto?

- Asignado la responsabilidad de un objeto intermedio que medie entre otros componentes o servicios, de tal manera que los objetos no están directamente acoplados. El objeto intermedio crea una indirección entre los componentes.

10. Principios generales de diseño

Composición sobre herencia

La herencia supone un acoplamiento muy fuerte que puede ocasionar problemas:

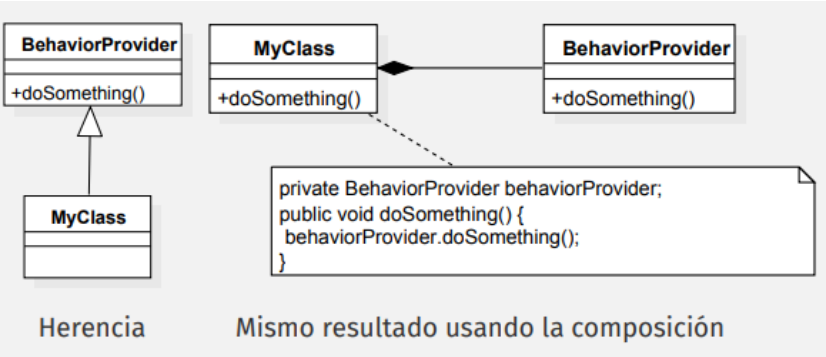
- Las relaciones “es un …” pueden dejar de cumplirse al incorporar nuevos cambios.
- Heredar de una clase para adquirir un comportamiento nos impide heredar de otra
- Los cambios de comportamiento en tiempo de ejecución son difíciles
- La herencia es difícil de implementar cuando los modelos persisten en una base de datos relacional.

Síntomas de que una herencia puede no ser adecuada:

- Numerosos métodos heredados que no se usan
- Demasiados métodos sobreescritos
- Demasiados niveles de herencia

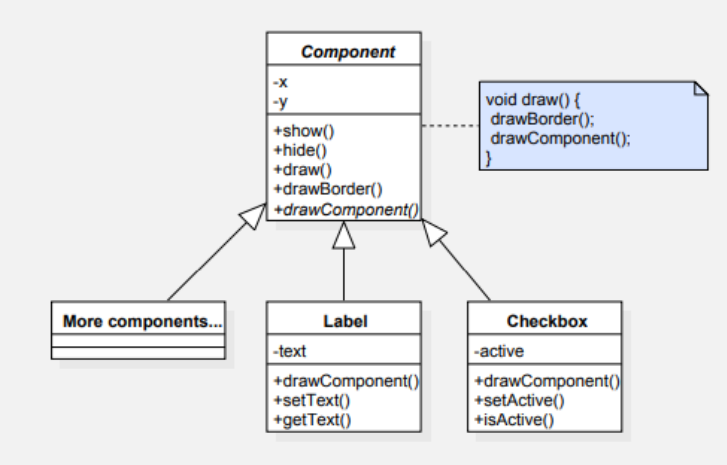
En general, la herencia sólo es imprescindible cuando necesitamos el polimorfismo. Aún así, en algunos casos se puede evitar la herencia simplificando el problema, a costa de sacrificar otros aspectos del diseño.

Otra forma de “heredar” un comportamiento es usando la composición, delegando la implementación de la responsabilidad a otra clase. Si estamos diseñando el modelo de dominio, la nueva clase no tiene por qué persistir en base de datos.



La composición ofrece mayor flexibilidad en tiempo de ejecución, ya que permite cambiar fácilmente el comportamiento de una clase.

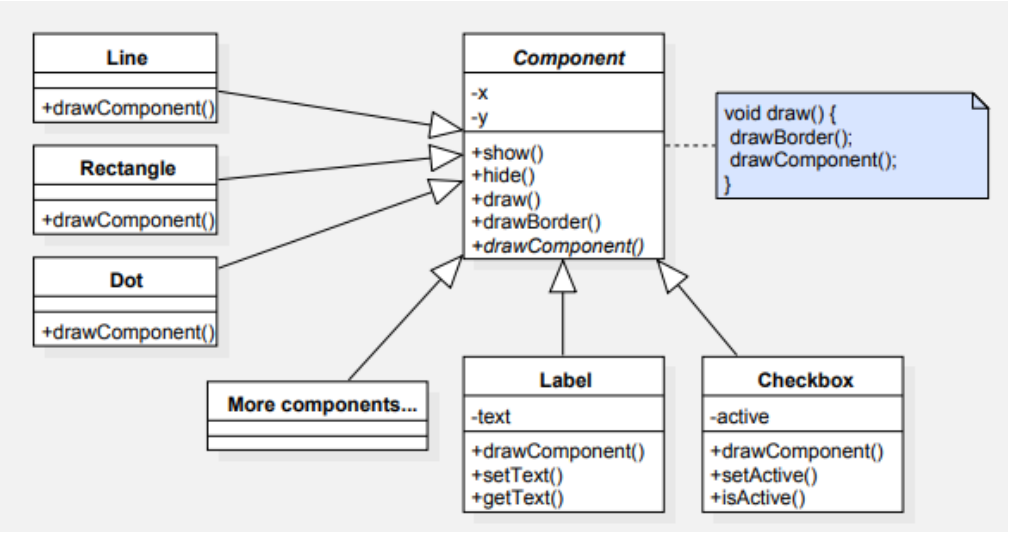
Supuesto: Añadimos la funcionalidad drawBorder() a la clase base Component.



Problemas de este diseño

- Baja la cohesión de la clase Componente
- Dificulta el mantenimiento, p.ej. si queremos añadir nuevos tipos de borde.

Supuesto: para enriquecer aún más la interfaz, nos piden añadir líneas, puntos y algunas formas geométricas para aumentar las posibilidades de diseño gráfico.

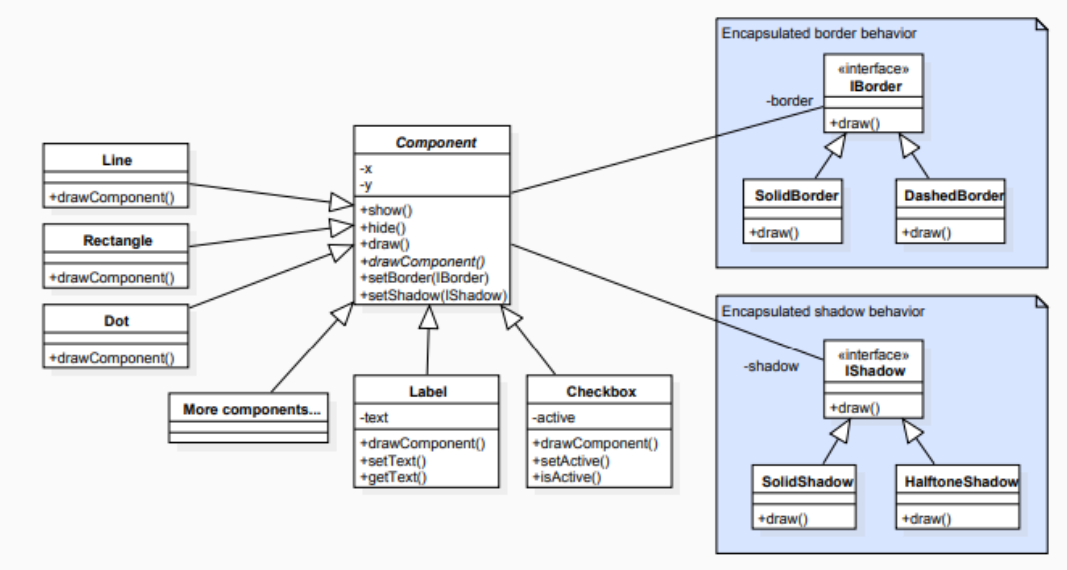


Problema

- La herencia provoca un comportamiento no deseado: los nuevos componentes también tiene bordes.

Solución

- Podemos solucionar este problema usando la composición en lugar de la herencia.



Principios SOLID

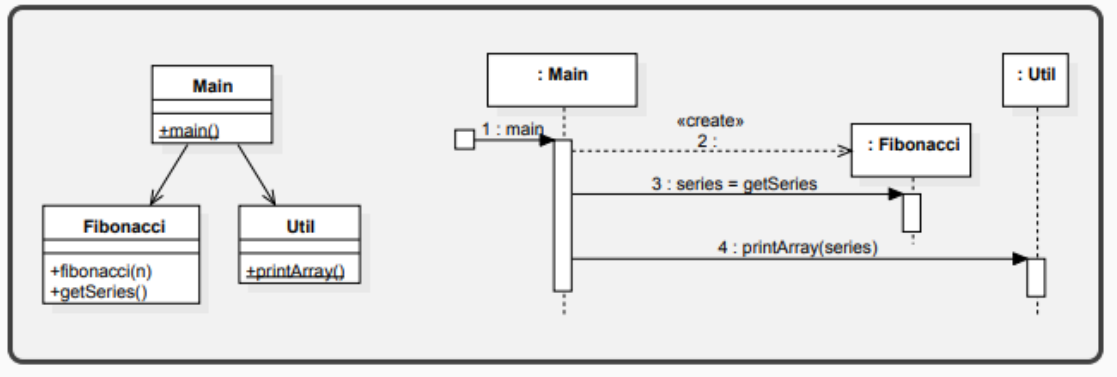
5 principios fundamentales de diseño [Martin, 2000]

- Principio de responsabilidad única (**S**ingle-responsability)
- Principio abierto/cerrado (**O**pen/Closed)
- Principio de sustitución de Liskov (**L**iskov substitution)
- Principio de segregación de interfaces (**I**nterface segregation)
- Principio de inversión de dependencias (**D**ependency inversion)

Supuesto: implementación de la clase Fibonacci del ejercicio de la primera sesión de prácticas. ¿Hay algún problema en este diseño?

```
class Fibonacci {
  private ArrayList<Integer> series
    = new ArrayList<Integer>() {{ add(0); add(1); }};

  public int fibonacci(int n) {
    int len = series.size();
    if (n > len) {
      for (; len<n; len++)
        series.add(series.get(len-1) + series.get(len-2));
      return series.get(len-1);
    }
    else
      return series.get(n-1);
  }
  public void printSeries() {
    System.out.println(Arrays.toString(series.toArray()));
  }
}
```

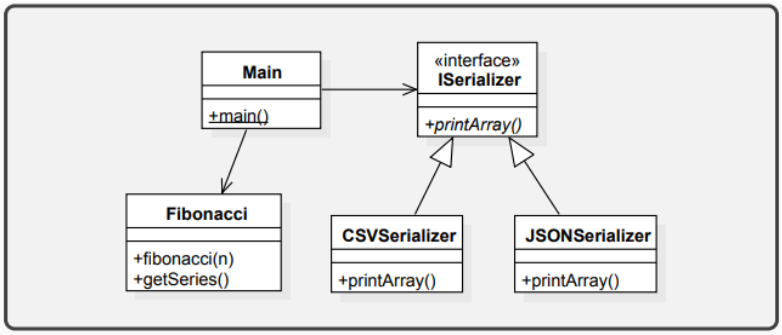
Principio de responsabilidad única: “Una clase sólo debería tener un motivo para cambiar”. En el ejemplo, la clase Fibonacci no debería ser la encargada de imprimir la secuencia. Una posible solución:



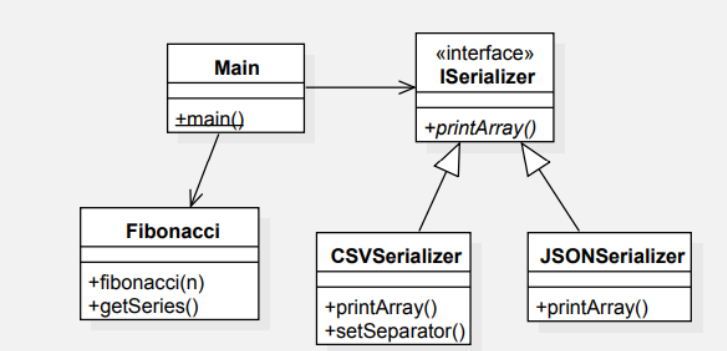
Supuesto: queremos imprimir un array usando distintos formatos (p.ej. CSV y JSON)

```
class Util {
  public void printArray(String format) {
    if (format.equals("csv")) {
      // Print as CSV
    } else if (format.equals("json")) {
      // Print as JSON }
  }
}
```

Principio abierto/cerrado: “Las entidades de software (clases, módulos, funciones, etc.) deberían estar abiertas para la extensión, pero cerradas a la modificación”. La forma tradicional de conseguirlo es mediante la herencia o implementación de interfaces.



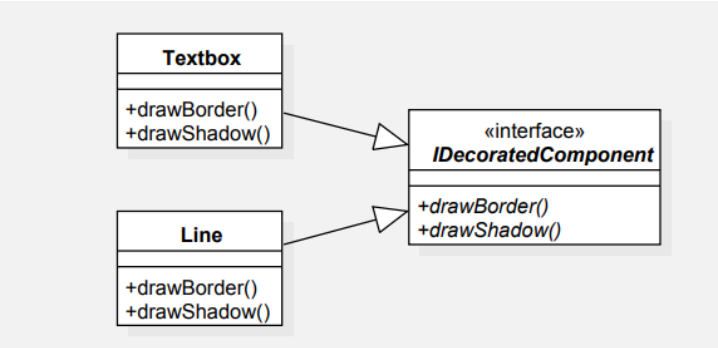
Supuesto: añadimos la posibilidad de especificar el tipo de separador para archivos CSV.



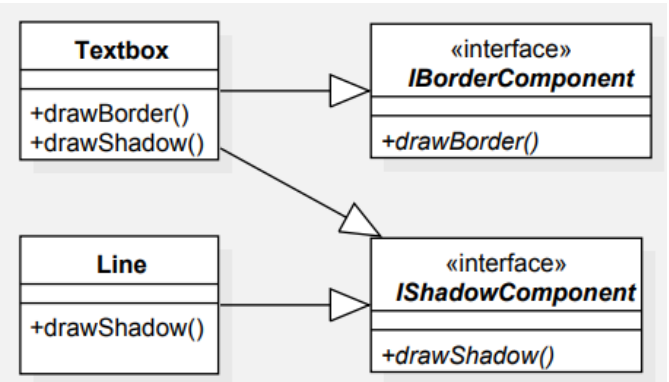
```java
class Main {
    public static void main(String args[]) {
        String format = args[1];
        ISerializer out = null;
        if (format.equals("csv")) {
            out = new CSVSerializer();
            ((CSVSerializer) out).setSeparator(":");
        }
        else if (format.equals("json"))
            out = new JSONSerializer();

        Fibonacci fib = new Fibonacci();
        // Computes the 10 first elements in the series
        fib.fibonacci(10);

        // Prints the array
        out.printArray(fib.getSeries());
    }
}
```

Principio de sustitución de Liskov: “Si S es un subtipo de T, entonces los objetos de tipo T en un programa deberían poder ser sustituidos por objetos de tipo S.” Al usar las subclases no deberíamos tener que conocer los detalles específicos de cada una de ellas, ni usarlas de forma distinta.

Supuesto: agrupamos las funcionalidades para dibujar bordes y sombras en un único interfaz IDecoratedComponent.



Principio de segregación de interfaces: “Ninguna clase debería depender de métodos que no usa.”



Supuesto: asumamos que queremos que la clase Fibonacci mantenga el método printArray. En su código instanciamos la clase necesaria para serializar dependiendo del formato de salida.

```
class Fibonacci {
    public void printArray(String format) {
        ISerializer out = null;
        if (format.equals("csv"))
            out = new CSVSerializer();
        else if (format.equals("json"))
            out = new JSONSerializer();

        // Prints the array
        out.printArray(series.toArray());
    }
}
```

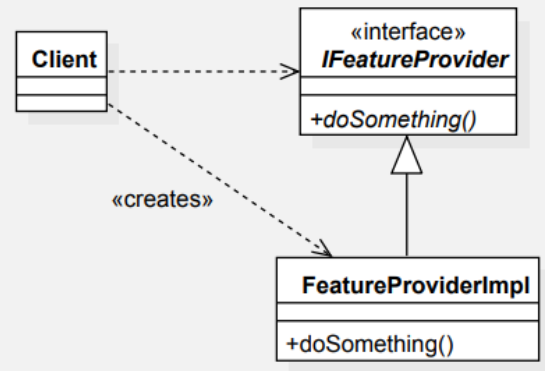
Principio de inversión de dependencias:

- “Módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones”
- “Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.”

En sistemas procedurales los módulos de alto nivel dependen de módulos de más bajo nivel. En sistemas orientados a objetos, los módulos de bajo nivel (implementaciones concretas) se consideran volátiles, ya que es más probable que cambien que las abstracciones. En la medida de lo posible se evitan estas dependencias ínvirtiéndolas, de manera que módulos de alto y bajo nivel dependen de las abstracciones. Es preferible el acoplamiento con el interfaz, antes que con las clases que lo implementan:

```
class Fibonacci {
    public void printArray(ISerializer serializer) {
        serializer.printArray(this.series.toArray());
    }
}

class Main {
    public static void main(String args[]) {
        ISerializer serializer = // Initialize depending on args
        Fibonacci fib = new Fibonacci();

        // do stuff...

        fib.printArray(serializer);
    }
}
```

Inyección de dependencias



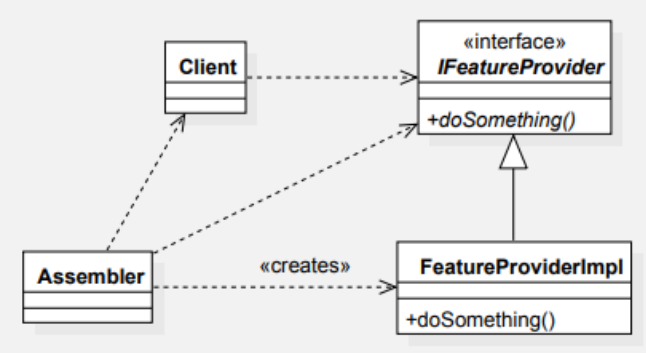
Problema:

- Cuando una clase A necesita una funcionalidad de otra clase B, y A crea una

instancia de B, se establece una dependencia en tiempo de compilación que no se puede cambiar en tiempo de ejecución.

Solución:

- En lugar de crear la instancia directamente, la clase recibe una instancia que implementa el interfaz de la funcionalidad que necesita. Esta técnica se conoce como **Inyección de Dependencias**. [Fowler, 2004]



La inyección de dependencias es una forma de inversión de control: un objeto distinto se encarga de crear la instancia de una implementación concreta y proporcionársela al objeto que la usará. Formas de inyección de dependencias:

- Inyección en el constructor
- Inyección como método setter
- Uso de un proveedor de servicios (Service Locator)

Inyección en el constructor

La clase cliente recibe una instancia del servicio en su constructor:

```
class Client {
  private IFeautreProvider provider = null;

  public Client(IFeatureProvider provider) {
    this.provider = provider;
  }
}
```

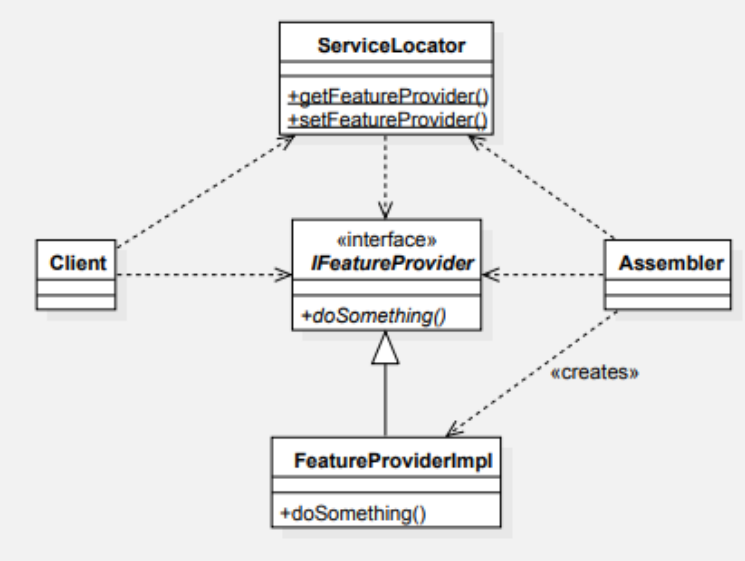
Inyección con método setter

La clase cliente recibe una instancia del servicio mediante un método setter:

```
class Client {
  private IFeautreProvider provider = null;

  public void setProvider(IFeatureProvider provider) {
    this.provider = provider;
  }
}
```

Uso de un proveedor de servicios

El objeto cliente obtiene los servicios de un ServiceLocator




```
class Assemble {
   public void init() {
      IFeatureProvider provider = new FeatureProviderImpl();
      ServiceLocator.setFeatureProvider(provider);
   }

class Client {
    public void method() {
      IFeatureProvider provider = ServiceLocator.getFeatureProvider();
   }
}
```

Al seguir usando un objeto *Assembler* para crear las instancias tenemos más flexibilidad:

- Podemos probar el ServiceLocator independientemente
- Podemos inyectarle clases mock o stub para probar el cliente

Inyección vs. Service Locator

- La inyección mediante constructor o método setter permite identificar mejor las dependencias, con un Service Locator hay que buscar sus llamadas en el código
- El Service Locator centraliza la inyección de dependencias en un único objeto, son más fáciles de gestionar.

Constructor vs. método setter

- La inyección mediante constructor permite crear objetos válidos desde la inicialización y proteger campos que no deben cambiar
- El método setter permite cambiar el comportamiento del objeto una vez inicializado.
- Lo importante es mantener la consistencia
- Se pueden proporcionar los dos mecanismos, no son excluyentes

Existen frameworks que permiten hacer la inyección de dependencias de forma automática, p.ej. Spring. La configuración se realiza mediante archivos XML.
Laravel realiza la inyección de dependencias automáticamente en algunos tipos de objetos, como p.ej. en los controladores. El *Service Container* se encarga de resolver las dependencias en los constructores. Para poder resolver las dependencias hay que

registrarlas mediante *ServiceProviders*.

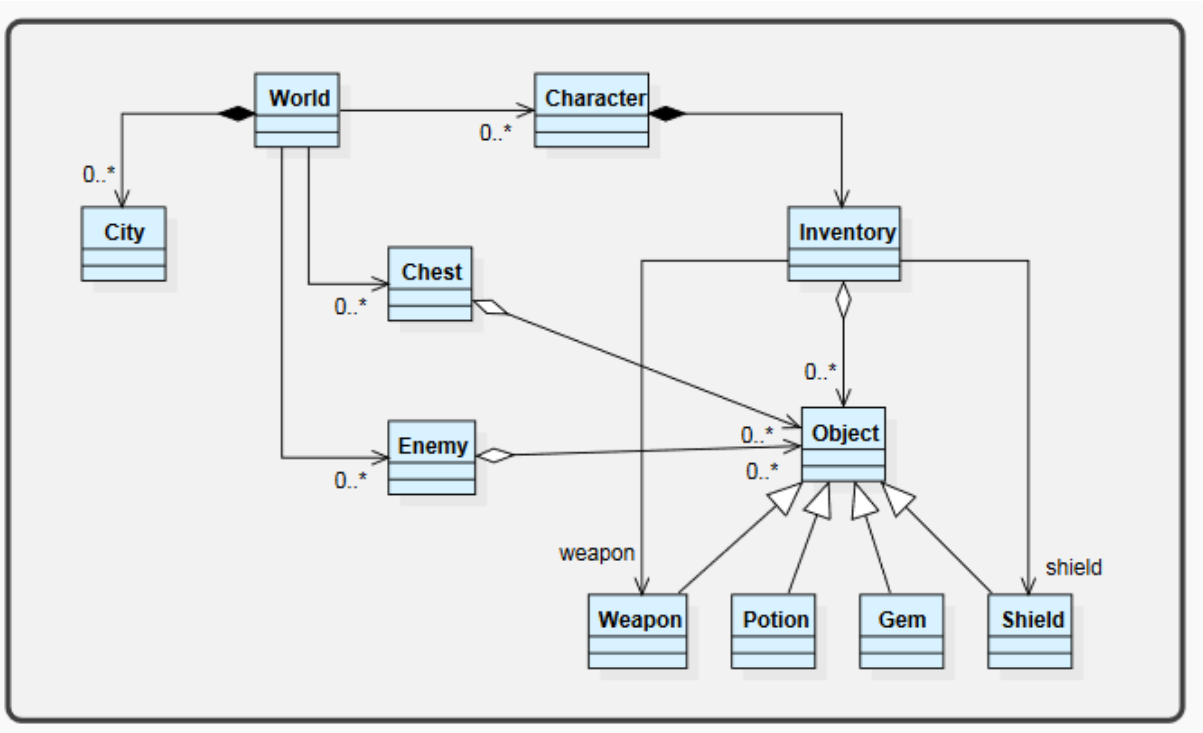
11. Introducción a los patrones GOF

Documentados por primera vez en [Gamma et al., 1994], es un catálogo de patrones de diseño detallado. Proporcionan soluciones a problemas cotidianos en el desarrollo de software. Se dividen en 3 tipos:

- **Creacionales**: Abstract factory, Builder, Factory method, Prototype, Singleton.
- **Estructurales**: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
- **De comportamiento**: Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor.

Diseña un diagrama de clases para un juego de rol con las siguientes características:

- Hay numerosos personajes que se mueven por un mundo abierto, en el que pueden encontrar distintas ciudades
- Los personajes tienen un inventario en el que guarda un arma, un escudo y los objetos que van consiguiendo.
- Se pueden conseguir objetos dentro de cofres o al derrotar a los pequeños enemigos que merodean por el mundo.
- Los objetos pueden ser pociones, gemas, pergaminos, etc.



Ampliación del juego:

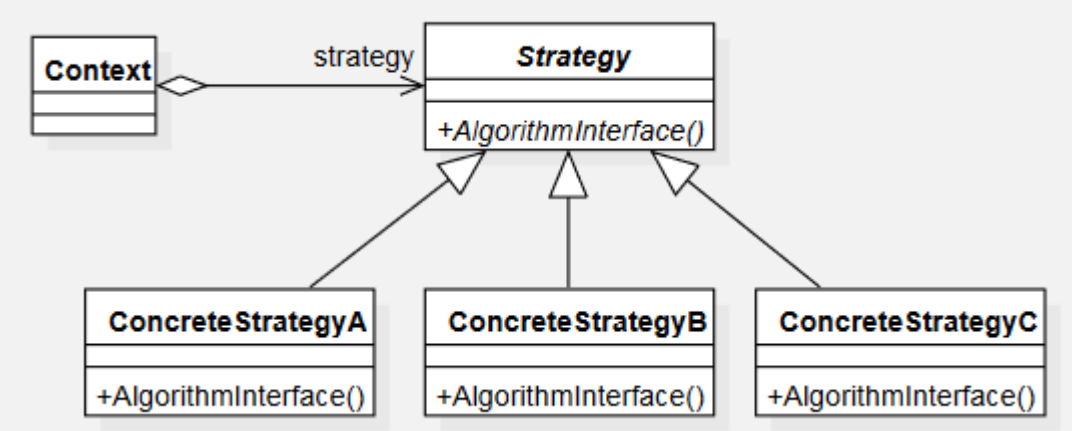
- En el juego los personajes podrán usar distintas armas, que se diferencian en los puntos de ataque (fuerza) que tienen
- En cada ataque, los personajes pueden elegir entre varios tipos de ataque: golpear,

empalar o ataques especiales; ya que cada ataque afecta de forma distinta a cada enemigo.

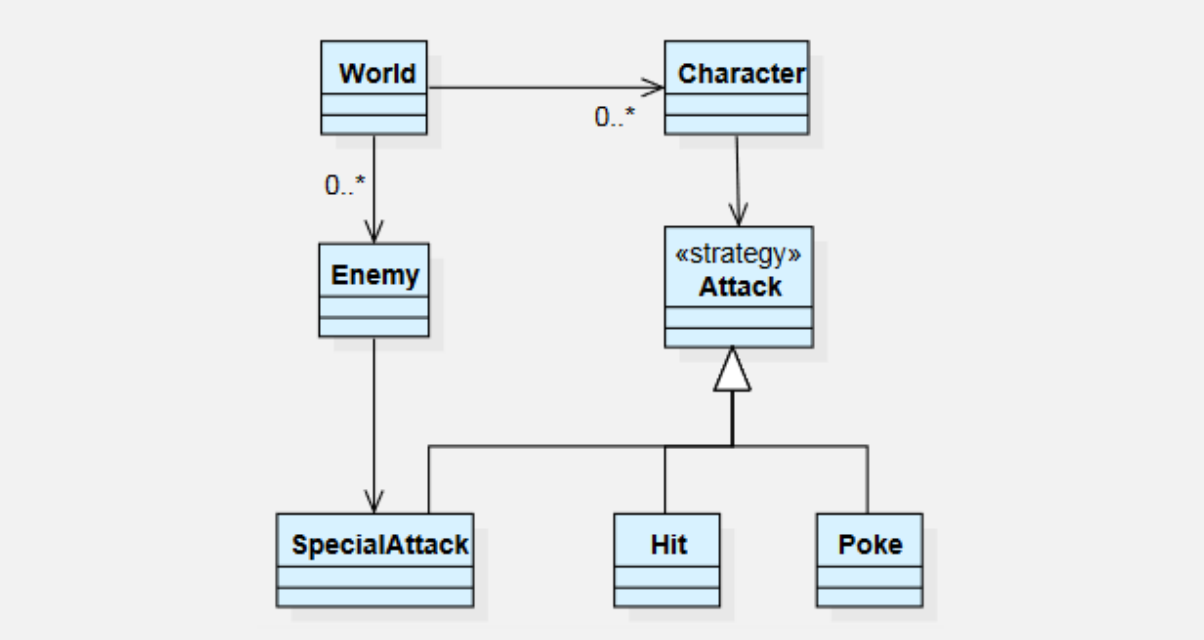
- Los personajes sólo podrán emplear los ataques especiales cuando los hayan obtenido al derrotar enemigos.

Patrón Strategy (de comportamiento)

Strategy: Define una jerarquía de clases que representan algoritmos, los cuales son intercambiables. Estos algoritmos pueden ser intercambiados por la aplicación en tiempo de ejecución.



Solución



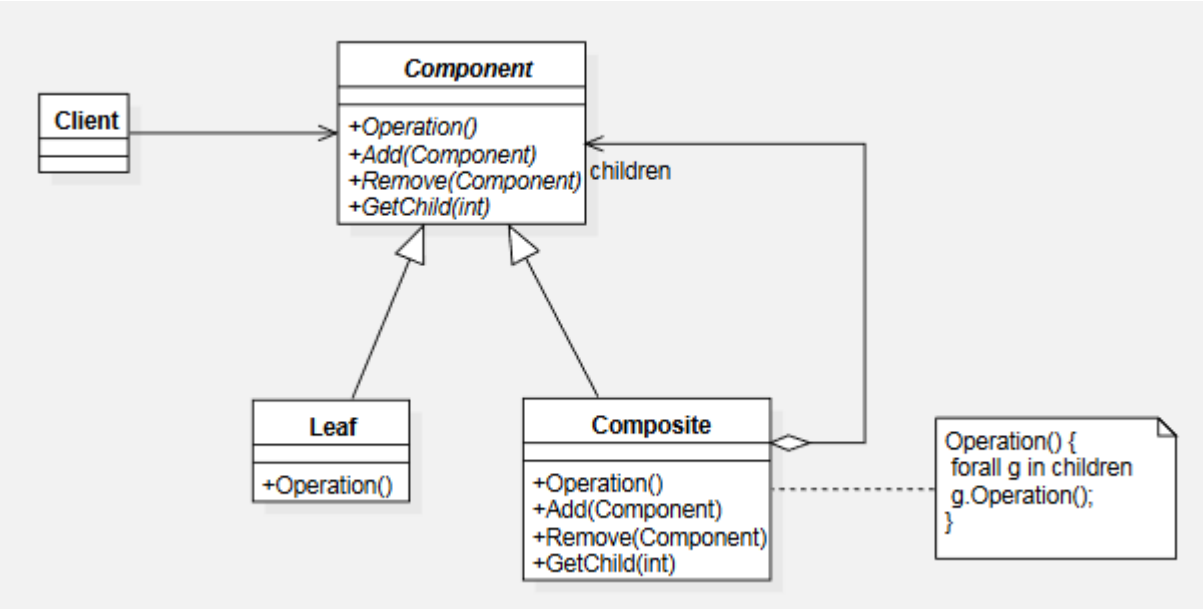
Aplicación de un patrón estructural

Ampliación del juego:

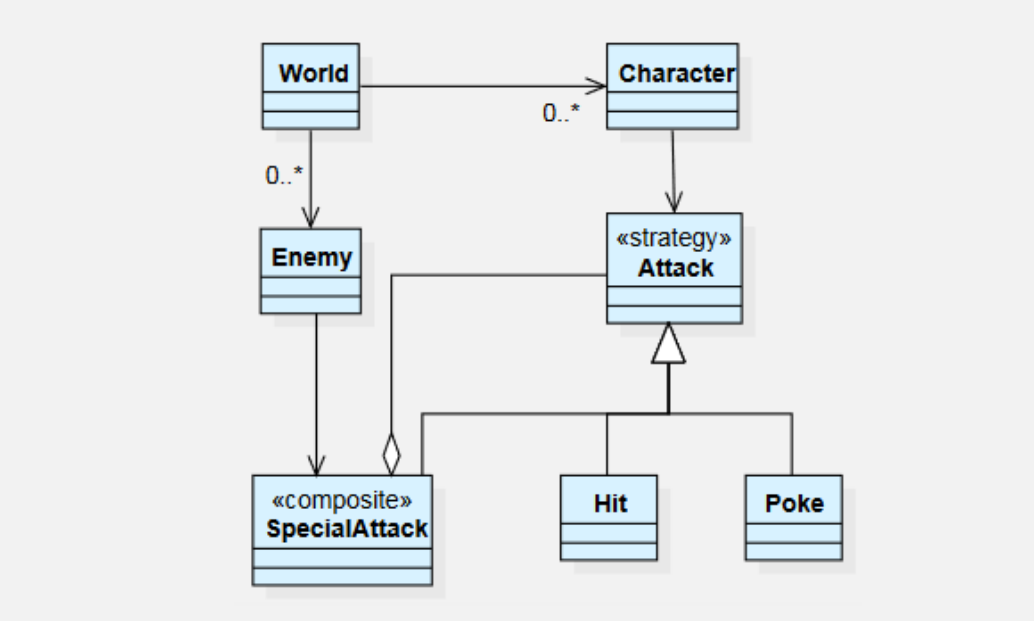
- Algunos ataques especiales consisten en combos: una combinación de otros ataques que el personaje ejecutará en secuencia.

Patrón Composite (estructural)

Composite: Compone objetos en estructuras de árboles para representar jerarquías parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los complejos.



Solución



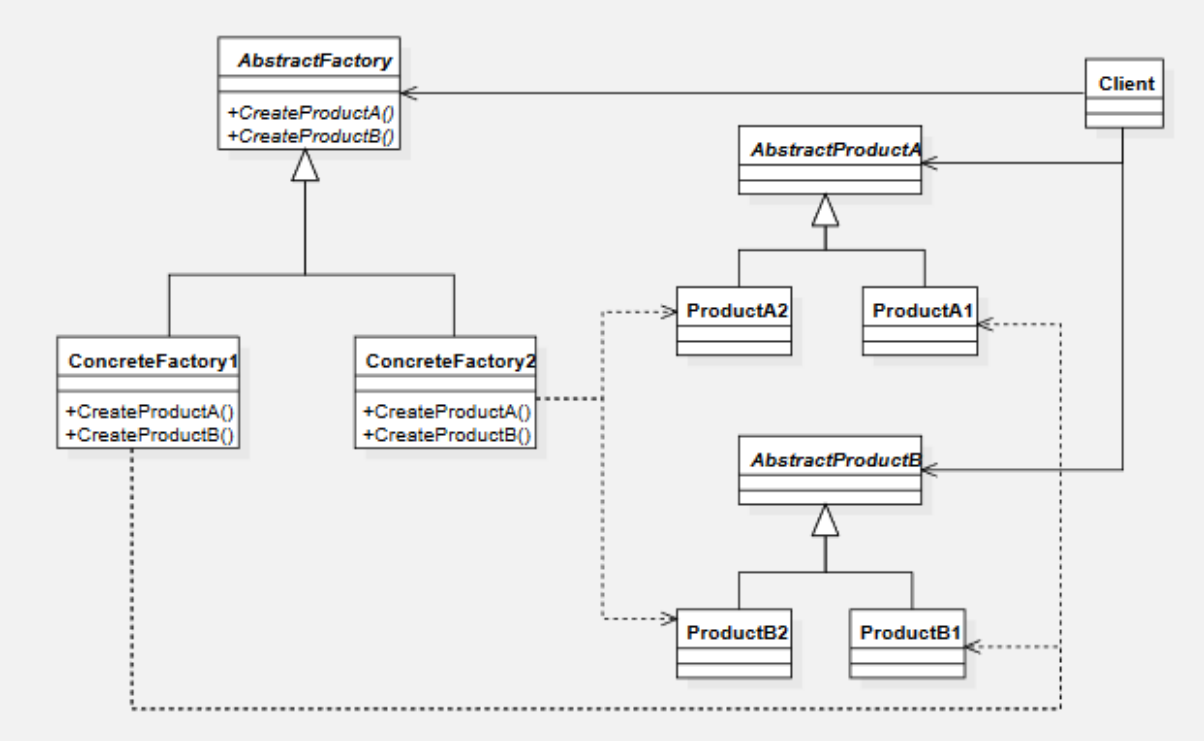
Ampliación del juego

- Queremos que las ciudades se generen automáticamente a medida que los personajes recorren el mundo abierto. La clase CityGenerator se encargará de crearlas.
- Las ciudades están compuestas de distintos tipos de edificios: posadas, herreros, tiendas, etc.
- Los edificios son distintos en función de la raza a la que pertenecen. En el juego

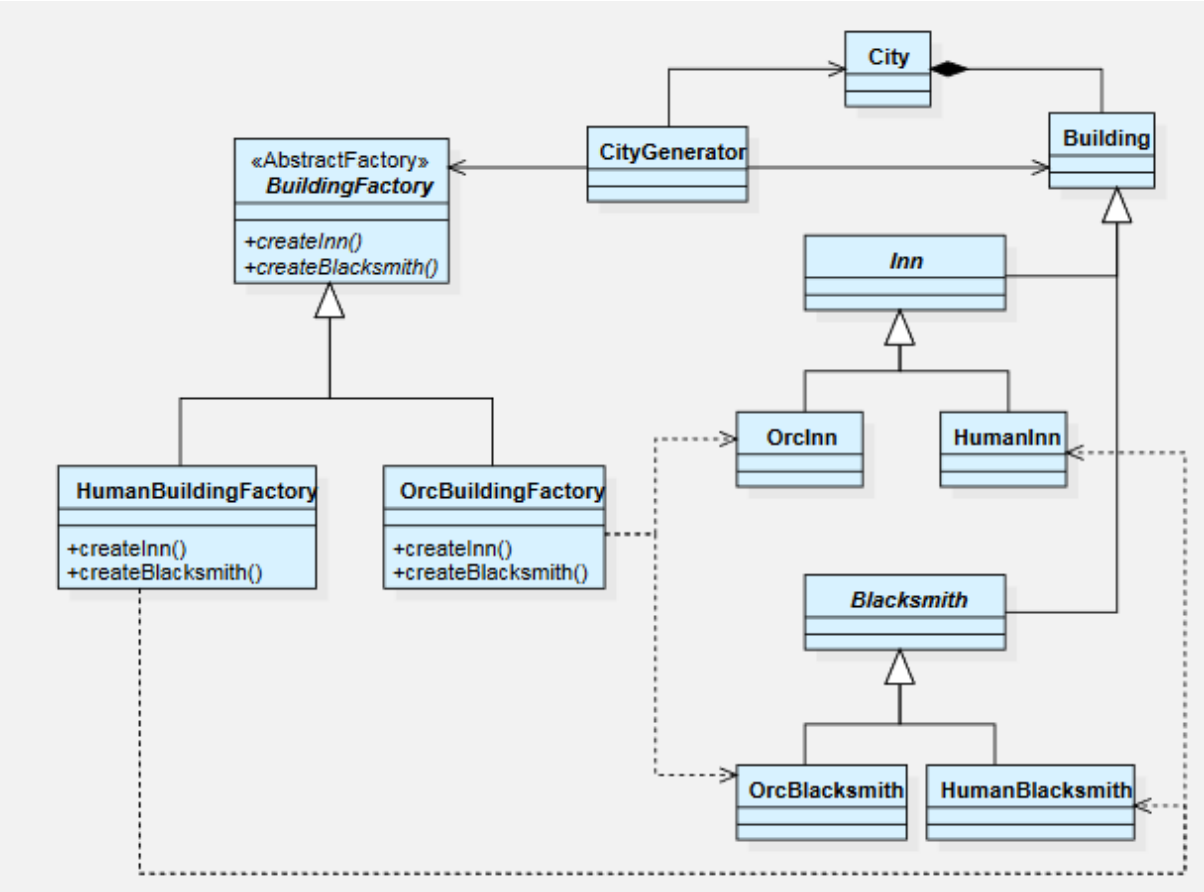
existen dos razas distintas, humanos y orcos, aunque se contempla que podrá haber más en el futuro.

- En ningún caso se podrán mezclar edificios de razas distintas en una misma ciudad.

Abstract factory: Provee una interfaz para crear familias de objetos “producto” relacionados o que dependen entre sí, sin especificar sus clases concretas.



Solución


City
CityGenerator
Building
«AbstractFactory»
BuildingFactory
+createInn()
+createBlacksmith()
Inn
OrcInn
HumanInn
HumanBuildingFactory
+createInn()
+createBlacksmith()
OrcBuildingFactory
+createInn()
+createBlacksmith()
Blacksmith
OrcBlacksmith
HumanBlacksmith

Exámenes DSS

JULIO 15

1 ¿Qué mide el rendimiento (performance) de un sistema?

a) El tiempo de respuesta

b) El tiempo que el sistema está funcionando

c) El tiempo que el sistema funciona sin fallos

2 ¿Cuándo es recomendable usar el patrón Data Mapper?

a) Cuando el modelo de dominio es simple

b) Cuando el modelo de dominio es complejo

c) Cuando el sistema usa una base de datos orientada a objetos

3 ¿Cuál de las siguientes afirmaciones sobre el patrón Remote Facade es FALSA?

a) Las fachadas remotas no deben contener lógica de dominio

b) Las fachadas remotas no deben tener estado

c) Las fachadas remotas deben ofrecer un único método para acceder a todas las propiedades de un objeto

4 En el patrón Model-View-Controller, cuando hay una serie de operaciones comunes que se deben realizar para cada petición del usuario, es conveniente usar el patrón:

a) Page Controller

b) Front Controller

c) Application Controller

5. ¿Cuál de los siguientes patrones no se aplica a la capa de dominio?

a) Transaction Script

b) Table Module

c) Table Data Gateway

6. Al usar el patrón Transaction Script:

a) Cada procedimiento creado es independiente del resto de capas del sistema

b) Surgen dos tipos de procedimientos: los que acceden a la base de datos y los que se comunican con el resto de capas del sistema

c) Cada procedimiento representa una acción que el usuario puede ejecutar

7. En una arquitectura de 3 capas, ¿cómo se relacionan las capas de dominio y presentación?

a) La capa de dominio nunca debe depender de la capa de presentación

b) La capa de presentación nunca debe depender de la capa de dominio

c) Nunca deben estar relacionadas

8. Al usar el patrón Active Record, ¿dónde debe situarse el código para acceder a la base de datos?

a) En el controlador del sistema

b) En clases especializadas para el acceso a datos

c) En las clases (objetos) de dominio

9. ¿Cuál es la diferencia entre los patrones Table Module y Domain Model?
a) Table Module pertenece a la capa de acceso a datos, y Domain Model a la capa de lógica de dominio
b) Normalmente, con Table Module hay un objeto por cada tabla, mientras que con Domain Model hay un objeto por cada fila de la tabla
c) Las dos son ciertas.
10. ¿En qué capa se deben situar las operaciones complejas?
a) En la capa de dominio
b) En la capa de servicio
c) En la capa de presentación

11. ¿Qué patrón podemos usar para mantener la consistencia cuando se modifican dos instancias de un mismo objeto desde distintas partes del sistema?
a) Unit of Work
b) Identity Map
c) Lazy Load

12. ¿Cuál de las siguientes responsabilidades NO corresponden a la capa de presentación?
a) Comunicarse con capas superiores para ofrecer funcionalidades complejas
b) Gestionar la interacción del usuario
c) Comunicarse con la capa de dominio para notificar los datos modificados por el usuario

13. En una arquitectura en capas, ¿qué capas pueden verse afectadas por una nueva funcionalidad?
a) Solamente la capa de dominio
b) Solamente la capa de presentación
c) Puede haber varias capas afectadas

14. El uso de interfaces de grano fino mejora el rendimiento de los sistemas distribuidos
a) Verdadero
b) Falso
c) Sólo si el diseño es orientado a objetos

15. ¿Con qué patrón de lógica de dominio se combina normalmente el patrón Row Data Gateway?
a) Table Data Gateway
b) Domain Model
c) Transaction Script

16. Para que un diseño de clases sea más fácil de entender y usar, la cohesión debe ser:
a) Baja
b) Alta
c) La cohesión no influye

17. ¿Qué patrón está pensado para poder intercambiar distintos comportamientos en tiempo de ejecución?
a) Strategy
b) Proxy
c) Builder

18. ¿Cuál de los siguientes patrones NO es de comportamiento?
a) Observer
b) Proxy
c) Command

19. El uso del patrón GRASP Creador implica:
a) Disminuye el acoplamiento entre clases
b) Aumenta el acoplamiento entre clases
c) No afecta al acoplamiento entre clases

20. ¿Qué inconveniente tiene el uso del patrón Composite?
a) El cliente no distingue entre clases simples y compuesta
b) No permite añadir nuevas clases simples
c) Resulta complicado imponer restricciones sobre la estructura de los objetos compuestos

21. ¿Qué inconveniente tiene el uso del patrón Abstract Factory?
a) Los productos de distintas familias comparten el mismo interfaz
b) No es fácil añadir nuevos productos
c) Para el cliente no es fácil seleccionar la familia de productos a crear

22. ¿Con qué patrón podemos reducir el acoplamiento entre dos partes de un sistema, cuando una parte usa un conjunto de clases de la otra?
a) Composite
b) Facade
c) Proxy

23. ¿Qué ventaja tiene el patrón Builder?
a) El director no necesita conocer los pasos del proceso de construcción
b) Cada constructor tiene un interfaz distinto
c) Permite crear distintos productos siguiendo el mismo proceso

24. ¿Cuál es la principal motivación de los patrones GRASP?
a) Crear modelos que no cambiarán a lo largo del proyecto
b) Proteger al sistema frente a posibles variaciones
c) Identificar las clases del modelo de dominio que se comunican con el resto del sistema

25. En el patrón Command, ¿qué información necesitan los comandos para ejecutarse?
a) Qué clase ha instanciado la acción
b) Qué clase ha invocado la acción

c) Qué clase es la receptora de la acción

26. ¿Qué patrón sería más adecuado para llevar un recuento del número de veces que se llama a cada método de un objeto?
a) Strategy
b) Observer
c) Proxy

27. En el patrón Observer, ¿qué rol es el encargado de llevar un registro de los objetos que deben ser notificados cuando hay un cambio?
a) Subject
b) Observer
c) Client

28. Cuál de los siguientes tipos de asociación implica un menor grado de acoplamiento
a) Uso
b) Herencia
c) Implementación de interfaces

29. El patrón GRASP indirección implica:
a) Relacionar dos clases mediante otra intermedia
b) Favorecer las relaciones directas entre clases
c) Asignar menos responsabilidades a las clases que hay en la frontera con otras capas

30. Para disminuir la dependencia entre clases y favorecer la reutilización de código el acoplamiento deber ser
a) Lo más bajo posible
b) Lo más alto posible
c) El acoplamiento no influye

JUNIO 14

Según el patrón GRASP Creador, la clase A debe ser la encargada de crear una instancia de la clase B si:
a) A contiene o agrega instancias de B

b) B usa instancias de A
c) Las dos son ciertas

¿Qué patrón GOF permite crear estructuras jerárquicas de objetos, de forma que el cliente pueda manejar objetos simples o compuestos indistintamente?
a) Builder
b) Strategy
c) Composite

¿Cuál es la diferencia entre los patrones de diseño y los frameworks?
a) Los Frameworks se basan en uno o varios patrones para solucionar problemas concretos
b) Los patrones usan frameworks para ofrecer soluciones reutilizables
c) Ninguna, los frameworks se usan para el diseño detallado de un sistema

El patrón GOF Abstract Factory:
a) Provee un interfaz para crear familias de productos de forma consistente
b) Permite controlar el número de instancias de los objetos que se crearán
c) Disminuye notablemente el número de clases del sistema

¿Cuál de las siguientes afirmaciones es CIERTA? Para que un diseño sea fácil de mantener y usar:
a) Debemos mantener un bajo acoplamiento y una baja cohesión
b) Debemos mantener un alto acoplamiento y una baja cohesión
c) Debemos mantener un bajo acoplamiento y una alta cohesión

¿Cuál de las siguientes afirmaciones sobre el patrón Remote Facade es FALSA?
a) Las fachadas remotas no deben contener lógica de dominio
b) Las fachadas remotas no deben tener estado
c) Las fachadas remotas deben ofrecer un único método para acceder a todas las propiedades de un objeto

Los objetos Data Transfer Object:
a) Solamente pueden contener datos de un objeto de dominio para garantizar la consistencia
b) Pueden contener datos de varios objetos de dominio para mejorar el rendimiento
c) No deben contener datos de los objetos de dominio para aumentar la cohesión

Cuando se desea simplificar el acceso a un subsistema o capa, ¿qué patrón GOF es el más indicado?
a) Strategy
b) Proxy
c) Facade

¿A qué nos referimos normalmente cuando hablamos de la arquitectura de un sistema?
a) A la infraestructura de hardware que dará soporte al sistema

b) A los principales componentes del sistema y sus relaciones
c) A la estructura de bajo nivel de cada componente del sistema

El acoplamiento entre clases es una medida de:
a) El grado de dependencia entre las clases del sistema
b) Cuantas funcionalidades distintas se asigna a cada clase
c) Cuantas clases intermedias hay que recorrer para llegar de una clase A a otra B

Las clases de diseño que surgen como resultado del patrón Fabricación Pura suelen aparecer:
a) Para representar objetos de dominio
b) Por descomposición funcional, para dividir responsabilidades
c) Por la aparición de jerarquías de herencia

¿Cuál es la principal ventaja de usar patrones GOF a la hora de diseñar un sistema?
a) Permite disminuir el número de clases del sistema
b) Hace que el sistema sea más fácil de comprender, a cambio de disminuir las posibilidades de reutilización de código
c) Aumenta la flexibilidad del sistema frente a futuros cambios

¿Cuáles son las capas típicas de un sistema de 3 capas?
a) Acceso a datos, servicio y lógica de dominio
b) Servicio, lógica de dominio y presentación
c) Acceso a datos, lógica de dominio y presentación

¿Qué problema pretende evitar el patrón Lazy Load?
a) Problemas de integridad referencial al cargar objetos relacionados
b) Problemas de rendimiento al cargar objetos relacionados
c) Problemas por el uso excesivo de herencia en la lógica de dominio

En el patrón GOF Observer, ¿cuál es la clase encargada de notificar un suceso en el sistema?
a) La clase que desempeña el rol Observer
b) La clase que desempeña el rol Concrete Observer
c) La clase que desempeña el rol Subject

JUNIO 16 - Modalidad 1

¿Cuándo conviene aplicar el patrón GRASP "Experto en Información" en cascada?
a. Cuando queremos prevenir la aparición de nuevas dependencias entre clases*
b. Cuando queremos romper intencionadamente la encapsulación de información.
c. Cuando queremos traspasar datos de la lógica de negocio a la capa de acceso a datos.

¿Qué tipo de acoplamiento debemos evitar para facilitar los cambios de

tecnología en
la capa de presentación?

a) Que la capa de presentación tenga como dependencias clases de la capa de lógica de negocio.

b) Que la capa de lógica de negocio tenga como dependencias clases de la capa de presentación.*

c) Que las clases de la capa de presentación tengan dependencias entre sí.

¿Qué ventaja NO podemos conseguir con el patrón GOF "Proxy"?

a. Controlar el acceso a un objeto.
b. Simular un objeto remoto de forma local.
c. Proporcionar una implementación alternativa para un objeto.*

¿Quién debe ser el encargado de crear los objetos de transferencia de datos (DTO)?

a. El objeto de dominio que contiene los datos.
b. Un objeto proxy que ocupa el lugar del DTO.
c. Un objeto Assembler que tiene acceso a los objetos de dominio.*

¿Qué inconveniente tiene el uso del patrón GOF "Composite"?

a. El cliente no distingue entre clases simples y compuestas.
b. No permite añadir nuevas clases simples.
c. Resulta complicado imponer restricciones sobre la estructura de los objetos compuestos.*

¿En qué se deben situar las operaciones complejas de una aplicación?

a. En la capa de dominio.
b. En la capa de servicio.*
c. En la capa de presentación.

Al usar el patrón Transaction Script

a. Cada procedimiento creado es independiente del resto de capas del sistema.
b. Surgen dos tipos de procedimientos: los que acceden a la base de datos y los que comunican con el resto de capas del sistema.
c. Cada procedimiento representa una acción que el usuario puede ejecutar.*

¿Con qué patrón podemos reducir el acoplamiento entre dos partes de un sistema, cuando una parte usa un conjunto de clases de la otra?

a. Composite
b. Façade*
c. Proxy

¿Con qué patrón podemos mantener la consistencia cuando se modifican dos instancias de un mismo objeto distintas partes del sistema?

a. Unit of work*
b. Identity Map

c. Lazy Load

¿Qué tipo de interfaces son deseables para llamadas entre objetos distribuidos?

a. Interfaces sin estado.
b. Interfaces de grano fino.
c. Interfaces de grano grueso.*

¿Qué inconveniente tiene el uso del patrón GOF "Abstract Factory"?

a. Los productos de distintas familias comparten el mismo interfaz.
b. No es fácil añadir nuevos productos.*
c. Para el cliente no es fácil seleccionar la familia de productos a crear.

¿Qué patrón es más recomendable para añadir nuevas funcionalidades a un objeto?

a. Decorator*
b. Composite
c. Adapter

Un lenguaje específico de dominio (DSL)…

a. Se construye a partir de un diagrama de clases UML.
b. Puede representar todo o una parte del dominio para el que está construido.*
c. Es menos abstracto que un lenguaje de propósito general.

¿Cuál es la diferencia entre los patrones Table Module y Domain Model?

a. Table Module pertenece a la capa de acceso a datos, y Domain Model a la capa de lógica de dominio.
b. Normalmente, con Table Module hay un objeto por cada tabla, mientras que con Domain Model hay un objeto por cada fila de la tabla.*
c. Las dos son ciertas.

¿Qué patrón permite reutilizar un mismo objeto de acceso a datos para distintas vistas?

a. Model View Controller
b. Model View Presenter*
c. Code-behind

¿En qué se basa el patrón GRASP "Polimorfismo"?

a. Las instancias de clases hijas se pueden comportar como si se tratase de la clase padre.*
b. Las instancias de una clase padre se pueden comportar como si se tratase de una clase hija
c. La introducción de una jerarquía de herencia disminuye el acoplamiento.

Los objetos de transferencia de datos (DTO)…

a. Se usan para pasar información entre distintas capas*
b. Contienen los métodos CRUD que se comunican con la base de datos.
c. Pueden contener tanto datos como métodos de lógica de negocio.

En una arquitectura de tipo Microkernel…

a. La escalabilidad es alta debido al pequeño tamaño del núcleo.
b. Cada plugin debe tener una interfaz independiente de las demás.
c. Se pueden añadir nuevas funcionalidades en tiempo de ejecución.*

En una arquitectura en capas abiertas…

a. Se permite que las capas inferiores se comuniquen con las superiores.
b. Las capas superiores pueden saltarse algunas de las capas inferiores.*
c. Se define una capa de servicio que puede ser usada opcionalmente por la capa de lógica de negocio.

¿Con qué patrón podemos recibir notificaciones cuando cambia el estado de un objeto?

a. Adapter
b. Observer*
c. Command

En el patrón GOF "Factory Method", la clase Creator proporciona una funcionalidad genérica independientemente del tipo de producto que se quiera crear.

a. Verdadero.*
b. Falso, esa responsabilidad corresponde a la clase ConcreteCreator.
c. Falso, esa responsabilidad corresponde a la clase ConcreteProduct.

¿En qué consiste el patrón GRASP "Indirección"?

a. Invertir el flujo de llamadas entre dos clases, facilitando así la automatización de pruebas.
b. Separar un conjunto grande de responsabilidades en dos clases distintas, favoreciendo así la cohesión.
c. Asignar una responsabilidad a una clase intermedia, desacoplando así dos clases del sistema.*

En el patrón GOF "Builder", ¿qué clases debe conocer la secuencia de pasos necesaria para construir un producto?

a. La clase Builder.
b. La clase ConcreteBuilder.
c. La clase Director.*

¿Qué beneficio obtenemos al usar patrones de software?

a. Disminuye la complejidad de los diseños.
b. Se reduce la cantidad de código que hay que crear.
c. Se simplifica la introducción de nuevas funcionalidades en el futuro.*

¿Cuál de los siguientes tipos de acoplamiento es más fuerte?

a. Cuando hay una jerarquía de herencia.*
b. Cuando una clase implementa un interfaz.
c. Cuando una clase recibe una lista de instancias de otra clase como parámetro en un método.

¿Con qué otro patrón GRASP está relacionado el patrón "Creador"?
a. Bajo acoplamiento, ya que disminuye el número de dependencias del sistema.*
b. Alta cohesión, ya que aumenta la cohesión de la clase creadora.
c. Controlador, ya que la clase creadora puede controlar a la clase creada.

¿Qué patrón permite centralizar en una sola clase las comprobaciones comunes en una implementación del patrón MVC (seguridad, personalización, etc.)?
a. Front Controller*
b. Page Controller
c. Application Controller

¿En qué se diferencian un modelo de dominio y un diagrama de diseño de clases?
a. El modelo de dominio se deriva a partir del diagrama de diseño de clases.
b. El diagrama de diseño de clases se deriva a partir del modelo del dominio.*
c. El modelo de dominio asigna responsabilidades a las clases, mientras que el diagrama de diseño de clases únicamente identifica las relaciones entre clases.

¿En qué caso está más indicado aplicar el patrón GRASP "Creador"?
a. Cuando queremos limitar el número de instancias de la clase creada.
b. Cuando la creación de instancias sigue una lógica condicional.
c. Cuando queremos almacenar instancias del objeto creado.*

¿Cuándo es necesario introducir una clase "Fabricación Pura"?
a. Cuando queremos una clase intermediaria para desacoplar dos clases ya existentes.
b. Cuando el aumento de responsabilidades de una clase pone en peligro su cohesión.*
c. Cuando necesitamos aplicar el patrón "Experto en información" en cascada.

JUNIO 17

¿Qué patrón GOF permite implementar de forma sencilla la funcionalidad "deshacer"?
a) Proxy
b) Command
c) Strategy

¿Qué patrón permite reutilizar un mismo objeto para proporcionar datos a distintas vistas?
A) Model View Controller
b) Model View Presenter
c) Code-behind

¿Cuándo es preferible usar una clase Factoría en lugar de aplicar el patrón GRASP Creador?
a) Cuando el tipo concreto del objeto a crear depende de un conjunto de condiciones
b) Cuando se debe inicializar el objeto en el momento de su creación *
c) Cuando el objeto creador debe almacenar los objetos creados

¿Qué problema se puede derivar de un excesivo acoplamiento en el diseño de un sistema?
a) Cada clase asume demasiadas responsabilidades
b) Los cambios en una clase pueden afectar a un gran número de clases distintas
c) Ninguno, si el acoplamiento no es entre clases de distintas capas

¿Qué tipo de acoplamiento debemos evitar para facilitar los cambios de tecnología en la capa de presentación?
a) Que la capa de presentación tenga como dependencias clases de la capa de lógica de negocio
b) Que la capa de lógica de negocio tenga como dependencias clases de la capa de presentación
c) Que las clases de la capa de presentación tengan dependencias entre sí

¿Qué inconvenientes tiene el uso del patrón GOF Composite?
a) El cliente no distingue entre clases simples y compuestas
b) No permite añadir nuevas clases simples
c) Resulta complicado imponer restricciones sobre la estructura de los objetos compuestos

¿Cuál de los siguientes tipos de acoplamiento es más fuerte?
a) Cuando hay una jerarquía de herencia *
b) Cuando una clase implementa una interfaz
c) Cuando una clase recibe una lista de instancias de otra clase como parámetro en un método

¿Qué patrón de lógica de negocios permite agrupar cada funcionalidad del sistema en un único método?
a) Domain Model
b) Table Data Gateway *
c) Transaption Script

¿Cuál es el principal problema del patrón “Class Table Inheritance”?
a) Es difícil usar un campo identificador único para todas las subclases
b) Las operaciones join necesarias pueden afectar al rendimiento*
c) Las columnas que no usan todas la subclases desperdician espacio en la base de datos *

¿Cuándo es necesario dividir una clase en dos o más clases distintas?
a) Cuando aplicamos el patrón “Experto en información” en cascada *
b) Cuando el acoplamiento de la clase es demasiado alto
c) Cuando la cohesión de la clase es demasiado baja

¿Con qué patrones GRASP está relacionado el patrón GOF Facade?
a) Controlador y bajo acoplamiento
b) Controlador y experto en información
c) Bajo acoplamiento y experto en información

¿Qué patrón GOF se usa para añadir nuevas funcionalidades a una clase sin modificarla?
a) Decorator*
b) Composite
c) Strategy

¿Qué información contiene un objeto Data Transfer Object?
a) Con qué Fachada Remota debe comunicarse el cliente para recuperar los datos
b) Reglas para transformar un objeto del modelo de dominio a representación textual
c) Datos de uno o más objetos del modelo de dominio *

¿Qué técnica de diseño de interfaces consiste en realizar un diseño separado de un sitio web para dispositivos móviles?
a) Adaptive web design *
b) Responsive web design
c) Device-oriented web design

¿Qué valor NO podemos conseguir con el patrón GOF Proxy?
a) Controlar el acceso a un objeto
b) Simular un objeto remoto de forma local
c) Proporcionar una implementación alternativa para un objeto *

¿En qué se basa el patrón GRASP Polimorfismo?
a) Las instancias de clases hijas se pueden comportar como si se tratase de la clase padre
b) las instancias de una clase padre se pueden comportar como si se tratase de una clase
c) La introducción de una jerarquía de herencia disminuye el acoplamiento

En el patrón GOF Builder, ¿qué clase debe conocer la secuencia de pasos necesario para construir un producto?
a) La clase Builder
b) La clase ConcreteBuilder
c) La clase Director

¿Qué patrón GOF nos permitirá saber cuántas veces hemos accedido a un objeto?
a) Proxy *
b) State
c) Strategy

¿En qué situación NO podemos reemplazar una jerarquía de herencia por una solución distinta en la composición?

a) Cuando la herencia se usa únicamente para heredar un comportamiento
b) Cuando necesitamos usar el polimorfismo
c) Cuando las clases hijas sobreescriben métodos de la clase padre

En el patrón State, ¿qué clase es la encargada de decidir cuál es el siguiente estado cuando hay un cambio de estado?
a) Solamente la clase Context puede tener esa información
b) Puede ser la clase Context o la clase ConcreteState
c) Ninguna de las clases implicadas en el patrón deberían tomar esa decisión *

¿Qué técnica podemos usar para evitar acoplar una clase con la implementación concreta de una funcionalidad?
a) Inversión de dependencias
b) Inyección de dependencias
c) Inversión de control

¿Quién debe ser el encargado de crear los objetos Data Transfer Object (DTO)?
a) El objeto dominio que contiene los datos
b) Un objeto proxy que ocupa el lugar del DTO
c) Un objeto Assembler que tiene acceso a los objetos de dominio *

¿En una arquitectura en capas, ¿por qué la capa de lógica de negocio se sitúa por encima de la capa de acceso de datos?
a) No es necesario, al ofrecer normalmente las mismas funcionalidades se puede intercambiar su orden*
b) Para evitar que la capa de acceso a datos acceda directamente a la capa de servicios
c) Para desacoplar a las capas superiores de los detalles de acceso a la base de datos

¿Qué patrón GOF permite limitar el número de instancias que se crean de una clase?
a) Template method
b) Factory method
c) Singleton*

Al implementar el patrón Observer, ¿qué clase es la encargada de notificar cuando hay novedades?
a) Subject
b) Observer
c) Subscription

¿Cuál de las siguientes NO es una ventaja de usar un framework arquitectural?
a) Podemos cambiar fácilmente la arquitectura del sistema
b) Proporciona una infraestructura que podemos extender con comportamiento personalizado
c) Establece las reglas mediante las que deben interactuar los componentes del sistema

Según el principio de inversión de dependencias...
a) Los módulos de bajo nivel no deben depender nunca de abstracciones
b) Los módulos de alto nivel no deben depender nunca de abstracciones

c) Los módulos de alto y bajo nivel deben depender de abstracciones

¿Qué objetos colaboran con el Front Controller para realizar comprobaciones de seguridad?

a) Middleware
b) Router
c) Application Controller

¿Con qué otro patrón GRASP está relacionado el patrón Creador?

a) Bajo acoplamiento, ya que disminuye el número de dependencias del sistema
b) Alta cohesión, ya que aumenta la cohesión de la clase creadora
c) Controlador, ya que la clase creadora puede controlar a la clase creada

En el patrón GOF Factory Method, la clase <<Creator>> proporciona una funcionalidad genérica independientemente del tipo de producto que se quiera crear

a) Verdadero
b) Falso, esa responsabilidad corresponde a la clase <<ConcreteCreator>>
c) Falso, esa responsabilidad corresponde a la clase <<ConcreteProduct>>