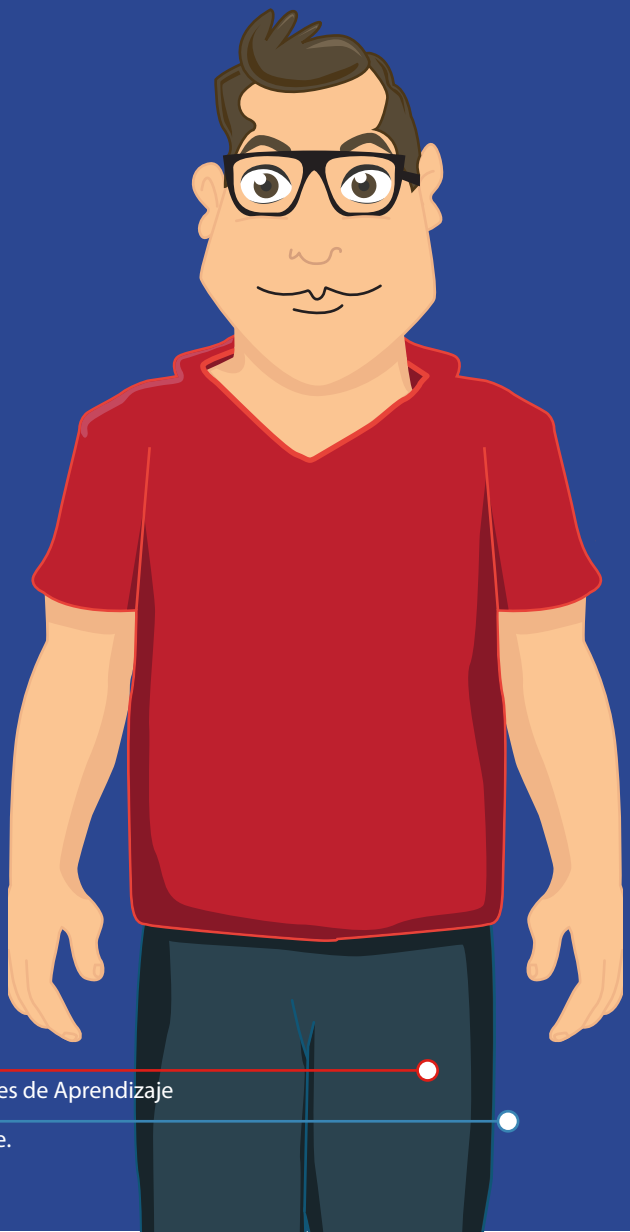
Patrones de Diseño



FAVA - Formación en Ambientes Virtuales de Aprendizaje

SENA - Servicio Nacional de Aprendizaje.

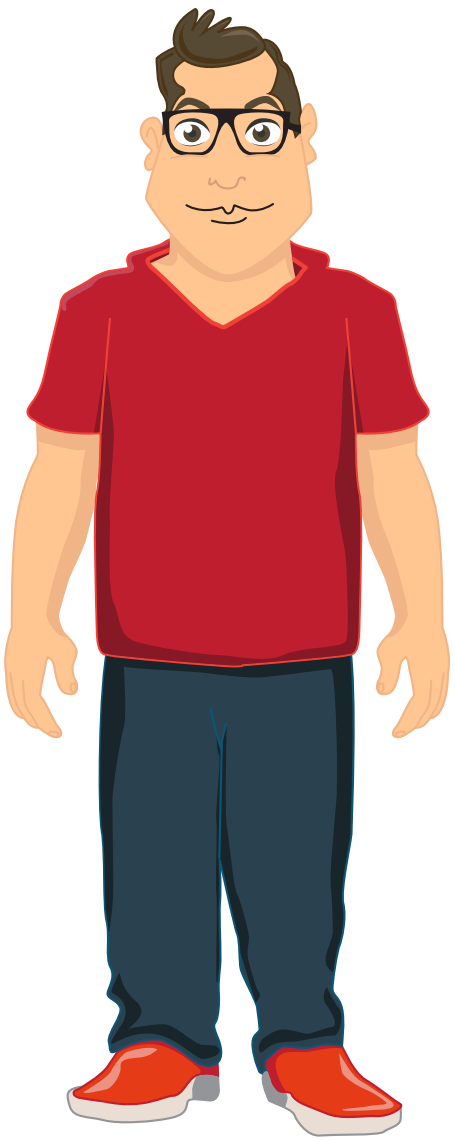
ESTRUCTURA DE CONTENIDOS

PATRONES DE DISEÑO

INTRODUCCIÓN

Durante la etapa de diseño de un sistema de información una de las principales tareas es la construcción de un prototipo con las clases que este sistema utilizará. En esta etapa existen problemas comunes al desarrollo de software que algunos autores han tipificados y generado soluciones que ha sido de gran aceptación. Las soluciones tipificadas, probadas y documentadas a estos problemas comunes se denominan patrones de diseño.

Los patrones de diseño son una herramienta que soporta la actividad de diseño de la arquitectura, proporcionando en algunas ocasiones el punto de inicio en la determinación de aspectos relacionados con la especificación de clases y su interacción en un sistema de información dado.

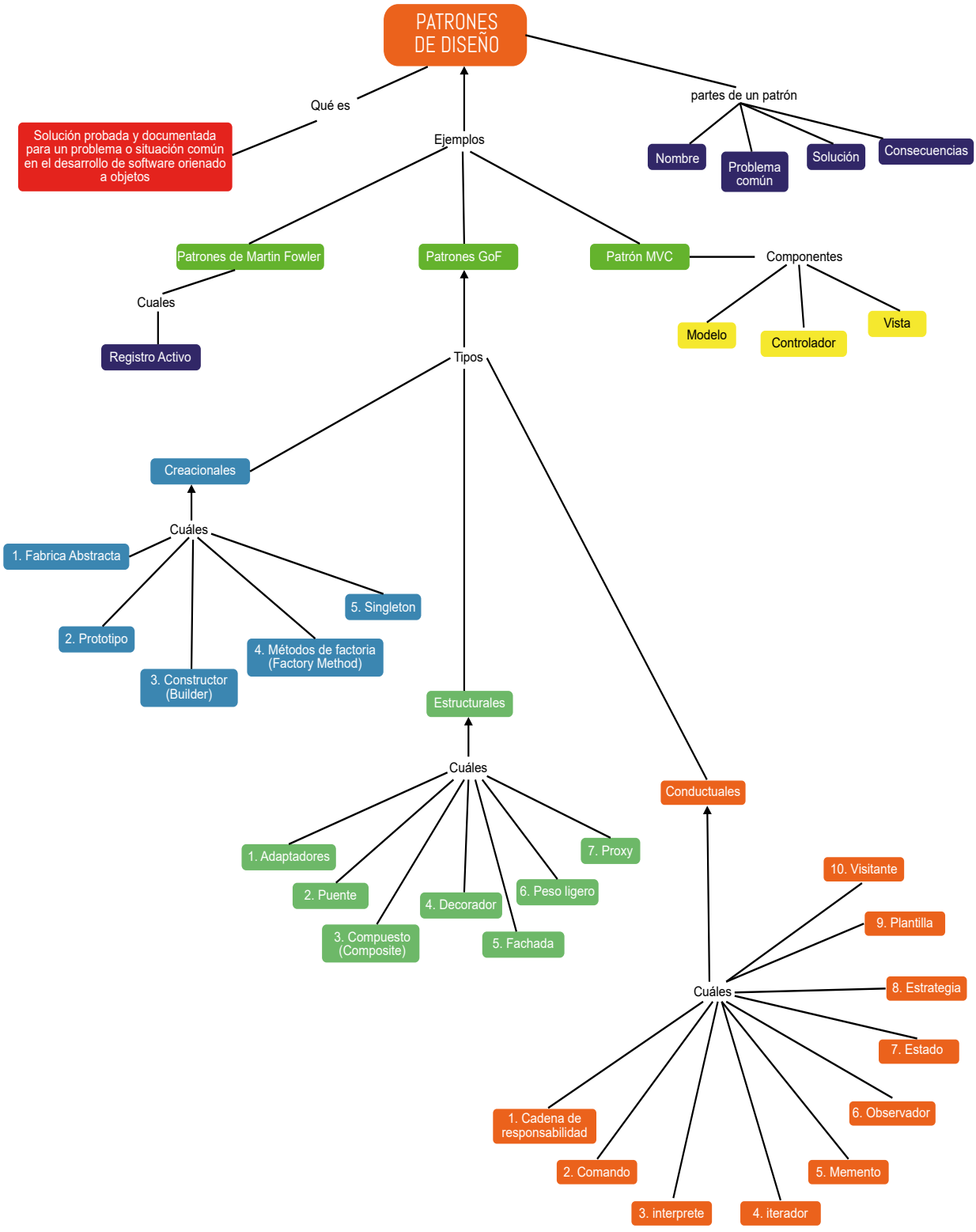


Es importante anotar que los patrones son influenciados por el paradigma de programación orientada a objetos. Desde este punto de vista los patrones o soluciones harán uso intensivo de las características propias de este paradigma como son: herencia, polimorfismo, encapsulamiento, entre otros.

Para un mejor entendimiento del recurso el aprendiz debe tener claro los siguientes conceptos de programación orientada a objetos vistos en el programa: clases, objetos, atributos, métodos, clase abstracta, interfaz, dependencia, generalización, especialización, instanciación,diagramas UML, entre otros.

La temática que se presenta en este objeto proporciona información sobre algunos de los patrones de diseño más importantes, con el fin de que sean considerados durante la fase de diseño del sistema de información que desarrollará el aprendiz.

MAPA DE CONTENIDO



DESARROLLO DE CONTENIDOS

1. Generalidades.

Un patrón de diseño es un modelo o plantilla de la solución a un problema común en el desarrollo de software. La historia de estos patrones se remonta a 1987 cuando Warde Cunningham y Kent Beck profesionales del desarrollo de software se dieron a la tarea de retomar algunas de las que consideraban “buenas ideas” en la solución de problemas orientados a objetos y desarrollaron cinco patrones de interacción publicados en un artículo académico (Fowler, 2002).

Sin embargo no fue hasta 1990 cuando el grupo denominado “Gang of Four” (La banda de los cuatro) que en adelante se identificarían como GoF retomaron un conjunto de 23 patrones de diseño y se empezó con mayor fuerza a difundir su uso.

Los patrones de diseño se enfocan en brindar una solución a problemas de software que han sido probados y tienen la documentación que permite analizar su posibilidad de uso. Esta documentación cuenta entre otros con (Gramma,1994):

- **Nombre:** describe el problema, solución y consecuencias a través de un nombre significativo.
- **Problema:** describe cuando debe utilizarse el patrón así como las condiciones para su aplicación.
- **Solución:** describe los elementos de diseño a incorporar, con la disposición general de las clases y objetos de forma abstracta.
- **Consecuencias:** los costos/beneficios al aplicar el diseño.

El principal objetivo que se consigue al utilizar esta técnica es el de la reutilización, lo que facilita conseguir apuntar a los principios de calidad en el diseño de software que son: la eficiencia, la mantenibilidad del sistema, que el sistema sea correcto y por ende la disminución en costos.

Una vez que se ha seleccionado el patrón de diseño a utilizar se recomienda leer inicialmente las partes relacionadas con la aplicabilidad y consecuencias para verificar que es el correcto, revisar concienzudamente la estructura para entender las clases y relaciones, así como verificar el código de ejemplo, que también puede ayudar a entender la implementación del mismo, escoger nombres significativos para los objetos y elementos, declarar las clases e identificar las clases que afectará, definir nombres para los métodos de acuerdo con la guía y por último redactar las operaciones.

2. Patrones GoF: Gang-of-Four.

Se pueden clasificar en 3 categorías (GRAMMA,1994):

- Patrones Creacionales (Creational patterns).
- Patrones Estructurales (Structural patterns).
- Patrones de Comportamiento (Bahavioral patterns).

2.1. Patrones creacionales.

Los patrones de creación muestran una guía de cómo crear objetos cuando su creación requiere tomar decisiones como qué clases instanciar o sobre que objetos se delegará responsabilidades.

Tienen relación con la instanciación de Clases. Pueden ser divididos en patrones de creación de clases (class-creation patterns) y patrones de creación de objetos (object-creational patterns).

- Los primeros usan la herencia dentro del proceso de instanciación.
- Los otros usan la delegación para realizar la tarea.

2.1.1. Patrón fabrica abstracta (abstract factory).

Problema:

¿Cómo crear familias de clases relacionadas que implementan una interfaz común? ¿Cómo hacer que el cambio de la familia de clases a utilizar sea transparente para la aplicación?

Solución:

Definir una interfaz que contenga métodos para la creación de los objetos de cada clase.

- Esta representa la Fábrica abstracta.
- Puede ser una interfaz o una clase abstracta.
- Definir las fábricas concretas para cada familia de clases.

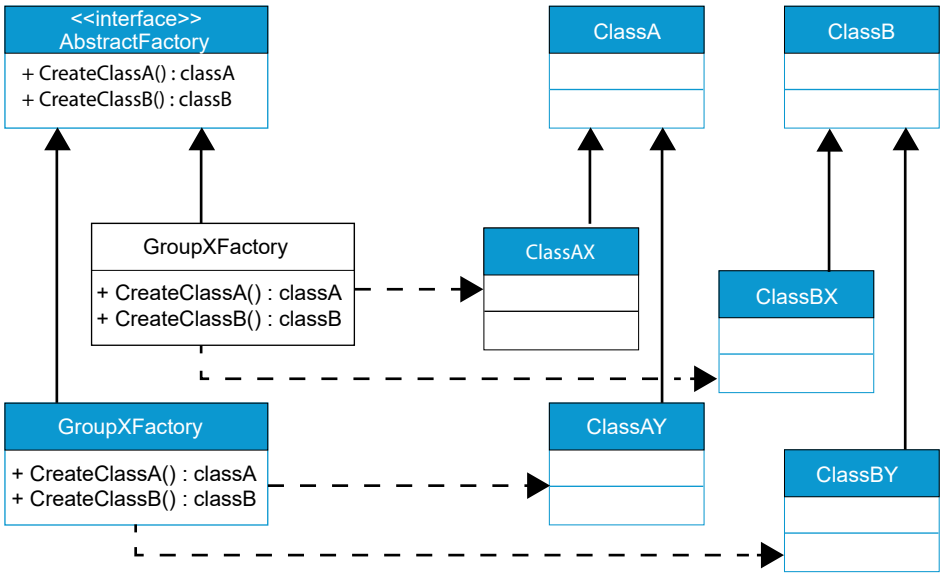


Figura 2.1. Patrón Fábrica Anstracta (Abstract Factory)

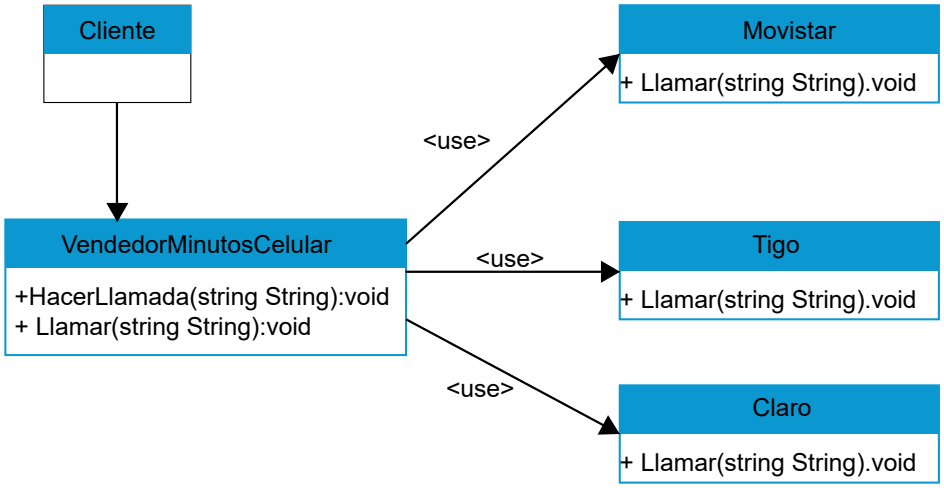


Figura 2.2. Ejemplo de Fábrica Abstracta (Abstract Factory)

2.1.2. Patrón Constructor (Builder).

Problema:

¿Cómo construir objetos complejos manteniendo la alta cohesión?
¿Cómo construir objetos que requieren varios pasos en su construcción?

Solución:

- Se crea una interfaz que defina cada uno de los pasos de la construcción.
- Se define una clase que dirija la construcción de los objetos.
- Para cada forma distinta de crear el objeto, se definine una clase concreta que la defina.

- Muy similar a Fábrica Abstracta.

Tener en cuenta que:

1. Fábrica Abstracta se aplica a familia de objetos.
2. Constructor se aplica a la creación objetos complejos

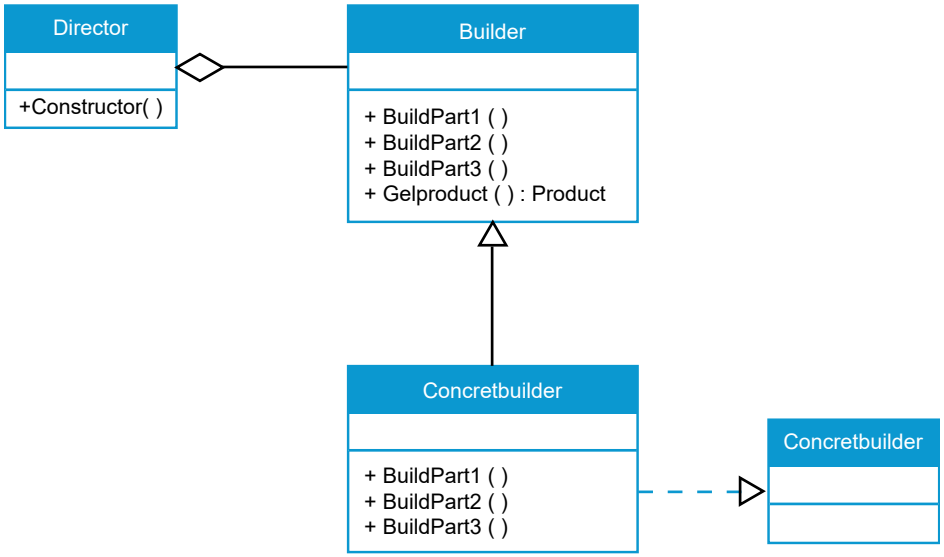


Figura 2.3. Patrón Constructor (Builder)

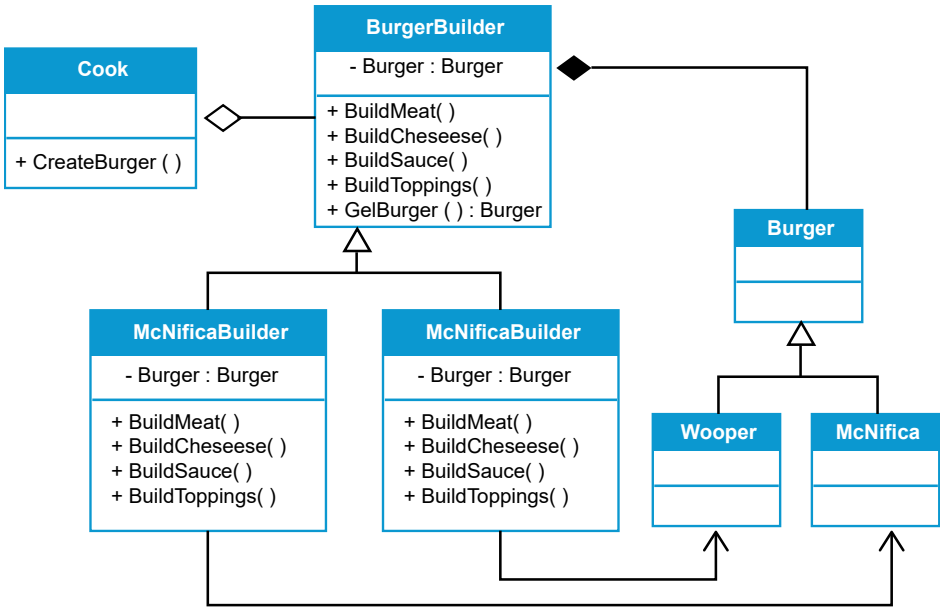


Figura 2.4. Ejemplo Patrón Constructor (Builder)

2.1.3. Patrón método factoría (factory method).

Problema:

¿Cómo crear instancias de elementos sin saber el tipo concreto del elemento?
¿Cómo crear instancias de una clase, donde la creación es un proceso complejo?

Solución:

- Se asigna la responsabilidad de la creación a un método particular.
- En caso de que sean múltiples tipos, es el método el que se encarga de la decisión de cuál elemento crear.
- En caso de creación compleja, el método encierra y se encarga de esta complejidad.

Tener en cuenta que:

1. Dado la responsabilidad de los constructores, estos no debieran dedicarse a lidiar con operaciones complejas.
2. Si existiesen operaciones muy complejas (lectura de archivos, acceso a bases de datos), en ese caso es mejor que un método se encargue de la responsabilidad de crear a los objetos de la clase.
3. Es una simplificación de Fábrica Abstracta, donde solo se construye instancias de una clase.

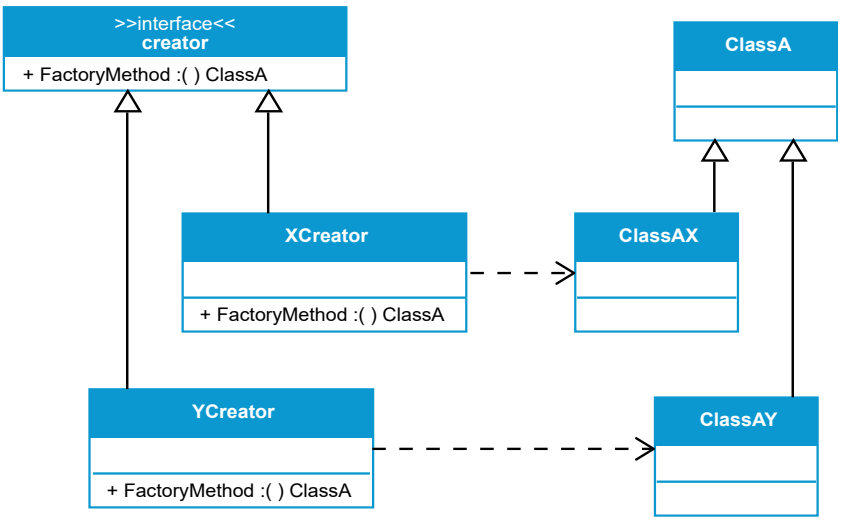


Figura 2.5. Patrón Método Factoría (Factory Method)

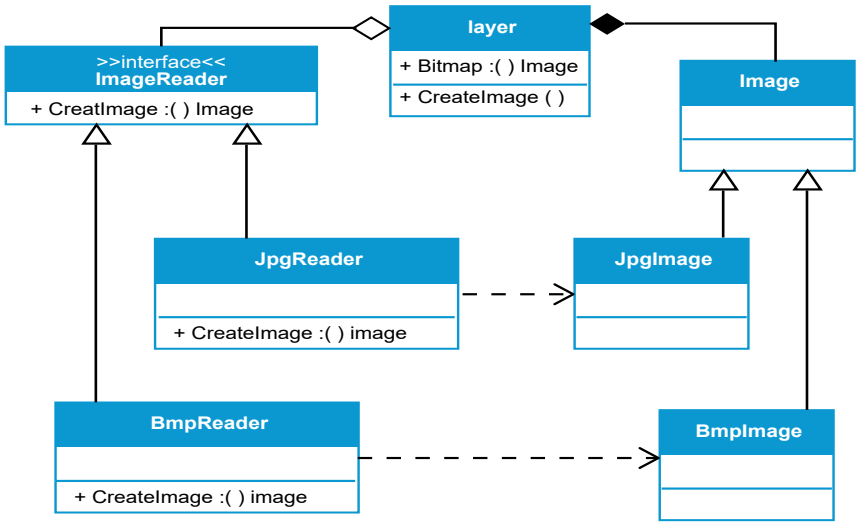


Figura 2.6. Ejemplo Patrón Método Factoría

2.1.4. Patrón prototipos (prototype).

Problema:

¿Cómo aumentar el rendimiento en la construcción de objetos que siempre son iguales?

Solución:

- Se crea una instancia del objeto y luego se clona.
- Para ello, se define una interfaz que debe implementar todos los objetos a ser clonados.
- En la mayoría de los lenguajes es posible hacer un Shadow copy de los objetos.
  - En C# existe el método MemberwiseClone().
  - En Java existe clone().
- Este se puede combinar con Método Factoría para la construcción de distintos objetos.

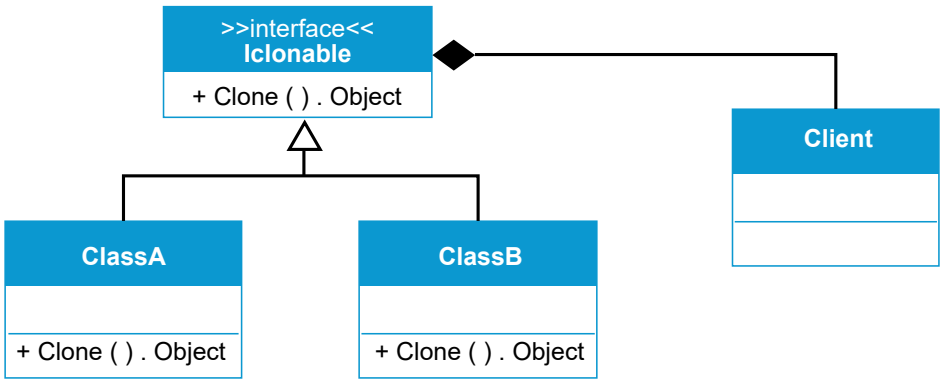


Figura 2.7. Patrón Prototipo

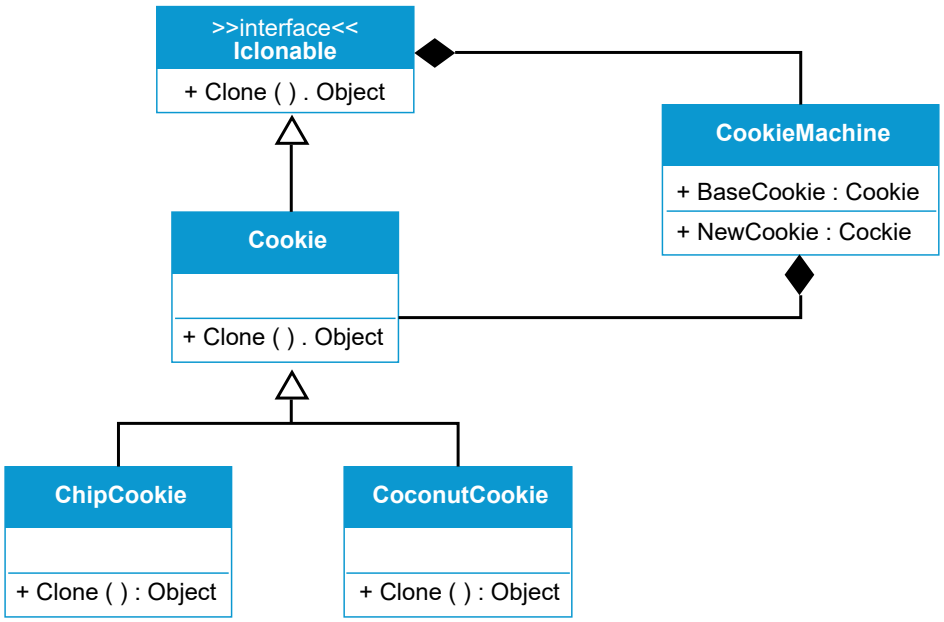


Figura 2.8. Ejemplo patrón prototipo

2.1.5. Patrón de diseño de instancia única (singleton).

Problema:
¿Qué se puede hacer para asegurar que de una clase determinada exista sólo una instancia durante toda la ejecución de la aplicación?

Solución:

- Dentro de la clase se elimina el acceso a los constructores, de modo que no sea posible construir elementos de esta clase, sólo a través de la misma clase. En la mayoría de los lenguajes esto se logra definiendo los métodos como private o protected.

- Se crea un método estático de acceso a esta instancia. Solamente la primera vez que se llame a este método se crea la instancia. Luego simplemente se retorna.

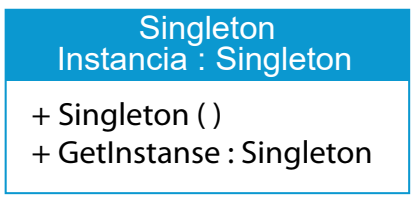


Figura 2.9. Patrón Singleton

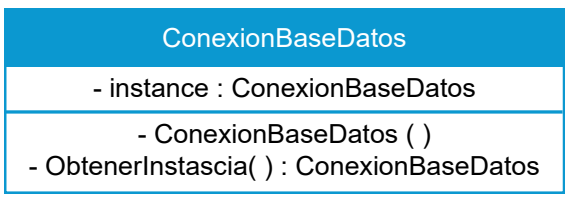


Figura 2.10. Ejemplo Patrón Singleton

2.2. Patrones estructurales.

Estos patrones describen las formas en que diferentes tipos de objetos pueden ser organizados para trabajar unos con otros, es decir la composición entre clases y objetos, se utiliza la herencia para componer interfaces y además definen formas de composición entre objetos para obtener determinada funcionalidad.

2.2.1. Patrón adaptador (adapter).

Problema:

¿Cómo hacer que una clase A utilice una clase B, si B no tiene la interfaz necesitada por A? Tener una Interfaz en su sentido más amplio. Tener métodos públicos.

¿Cómo hacer lo anterior sin cambiar B (porque no se tiene acceso, o porque está muy acoplada)?

Solución:

- Crear una clase que funcione de adaptador entre las dos clases que quieren comunicarse.
- El adaptador provee la interfaz que necesita la clase A.
- El adaptador traduce las llamadas a su interfaz en llamadas a la interfaz original (en este caso la clase B). En general se trata de transformaciones de parámetros. Pero puede llegar a cambiar las responsabilidades de la clase adaptada.
- Este patrón está pensado para situaciones en las que no se tiene acceso a la clase B, o se usa en otras partes del sistema y no se quiere modificar. Solamente la primera vez que se llame a este método se crea la instancia. Luego simplemente se retorna.

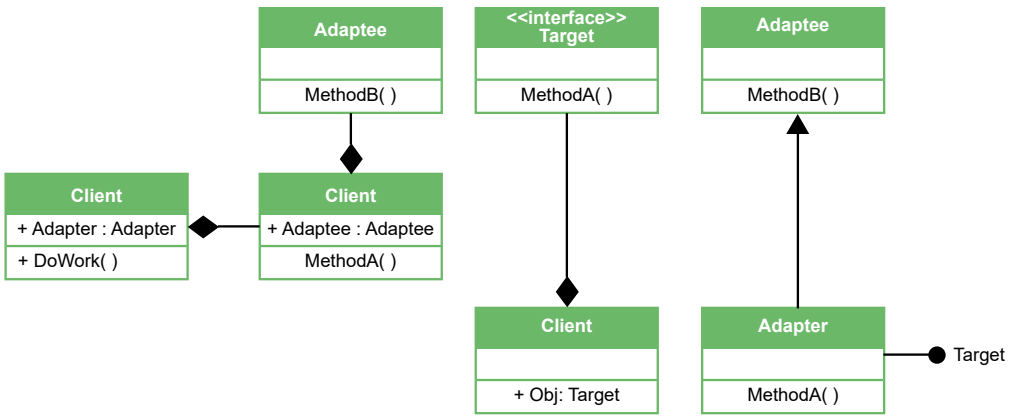


Figura 2.11. Patrón Adoptador

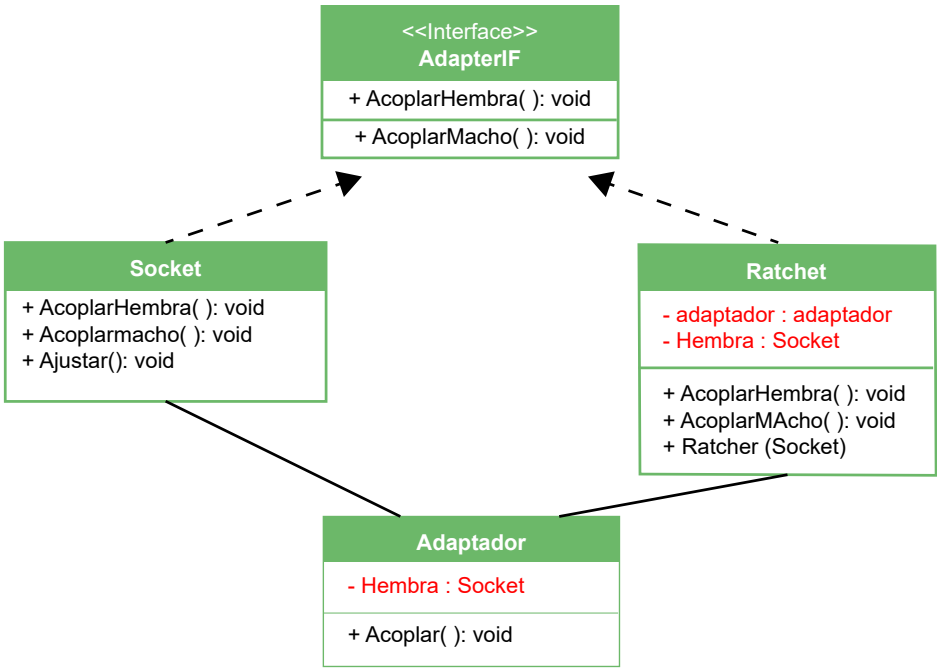


Figura 2.12. Ejemplo del patrón de diseño Adaptador

2.2.2 Patrón puente (bridge).

Problema:

¿Qué hacer para desacoplar una abstracción de su implementación de modo que ambas puedan variar independientemente? Abstracción se refiere a la interfaz.

¿Qué hacer cuando una implementación varía en tiempo de ejecución y se quiere mantener la abstracción estable?

¿Qué hacer cuando una abstracción varía en tiempo de ejecución y se quiere mantener la implementación estable?

Solución

- Se crea una clase abstracta que represente la abstracción y otra clase abstracta o interfaz que represente la implementación.
- La abstracción contiene un objeto de la implementación el cual utiliza para cumplir su propósito

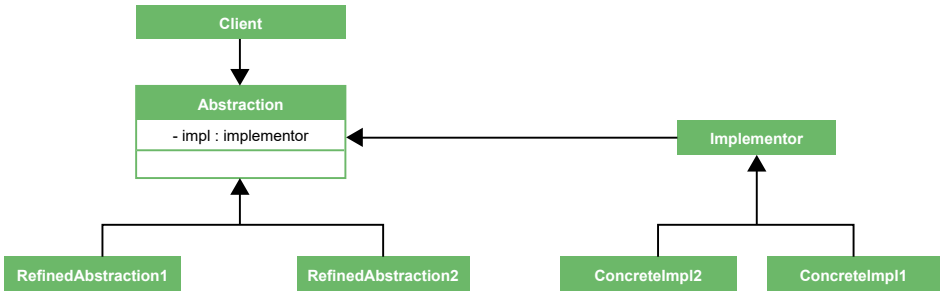


Figura 2.13. Patrón Puente (Bridge)

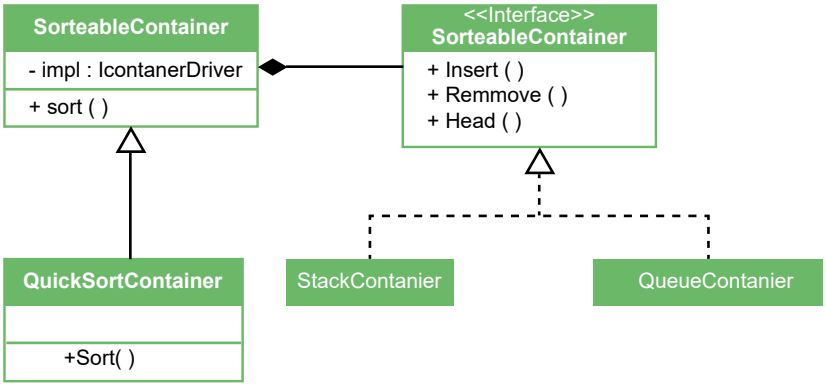


Figura 2.14. Ejemplo Patrón Puente (Bridge)

2.2.3 Patrón compuesto (composite).

Problema:

¿ Cómo tratar a un grupo de elementos de la misma forma como se trata uno de sus elementos? El grupo es una composición. El elemento único es un objeto atómico.
¿Cómo tratar a este mismo grupo si además permitimos que contenga otros grupos de los mismos elementos?

Solución:

- Definir clases para composiciones y objetos atómicos que implementen la misma interfaz.
- Los objetos se representan como una estructura de árbol.
  - Los objetos compuestos son ramas.
  - Los objetos atómicos son hojas.
- Permite la realización recursiva de las operaciones.

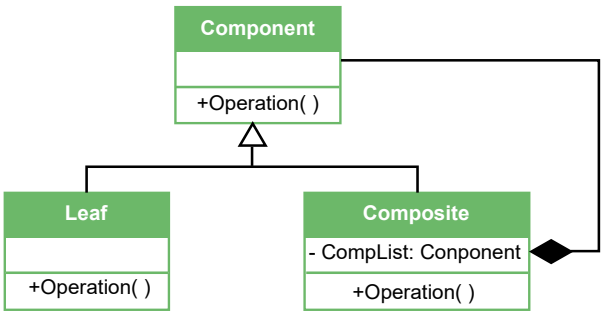


Figura 2.15 . Patrón Compuesto (Composite)

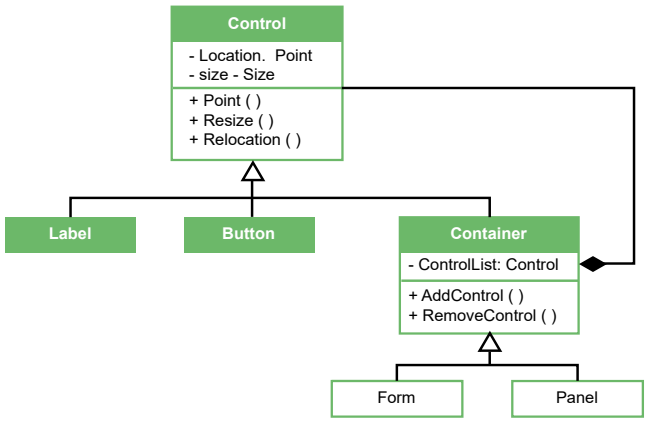


Figura 2.16 . Ejemplo Patrón Compuesto (Composite)

2.2.4. Patrón decorador (decorator).

Problema:

¿Cómo se pueden añadir nuevas funcionalidades a una clase de forma dinámica? ¿Cómo se puede decidir en tiempo de ejecución las funcionalidades que se desea que realice cierta clase? Herencia implica tener una subclase para cada combinación de funcionalidades. Para 4 funcionalidades serían 16 subclases distintas.

Solución:

- Se crea una clase que decore la clase a la cual se quiere agregar funcionalidad.
- La clase decoradora envuelve a la otra clase, pero comparte con esta la misma interfaz.

Se debe tener en cuenta que :

1. La clase decoradora tiene como atributo a la otra clase.
2. Como las dos clases comparten la interfaz, se puede tener una clase decoradora que envuelve a otra clase decoradora. De esta forma se puede elegir los adornos en tiempo de ejecución simplemente creando decoradores que envuelvan otros decoradores.

Funcionamiento recursivo similar a Compuesto.

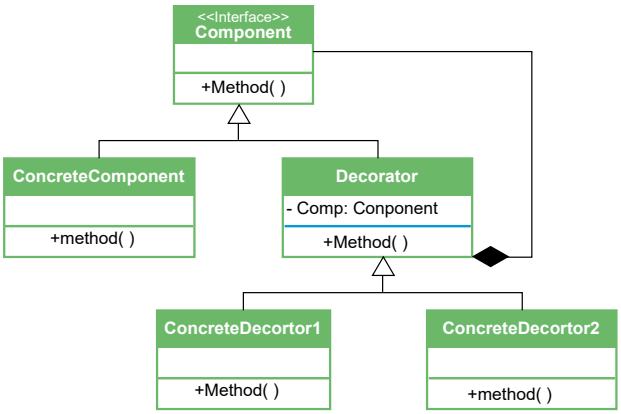


Figura 2.17. Patrón Decorador (Decorator)

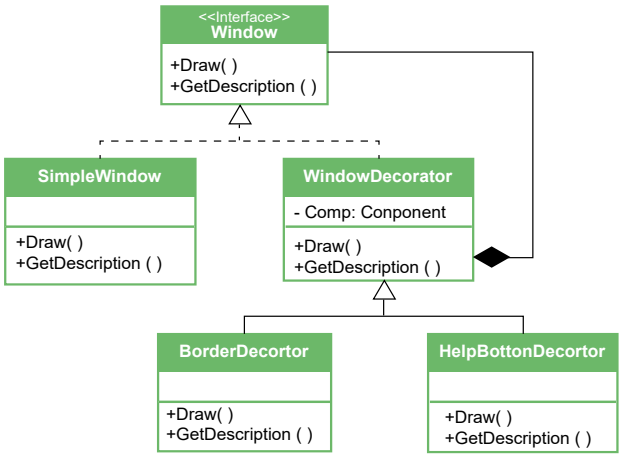


Figura 2.18. Ejemplo Patrón Decorador (Decorator)

2.2.5 Patrón fachada (facade).

Problema:

¿Cómo reducir el acoplamiento con un subsistema de clases complejas,que están sujetas a futuros cambios, pero manteniendo la funcionalidad del subsistema?

¿Cómo simplificar la realización de tareas comunes con una serie de clases de un subsistema?

Solución:

- Se crea una interfaz que exponga las funcionalidades del subsistema, es decir, un punto de contacto único con el subsistema, disminuyendo el posible acoplamiento con este. A esta interfaz se le llama la fachada del subsistema.
- La clase fachada se preocupa de trabajar con las clases del subsistema para conseguir la funcionalidad solicitada.
- Una fachada se usa para simplificar el trabajo con el subsistema.
- Es un patrón similar al patrón Adaptador. El Adaptador se utiliza cuando es necesario respetar una interfaz preexistente. Fachada se utiliza para definir una interfaz simplificada de acceso a un subsistema o conjunto de clases.

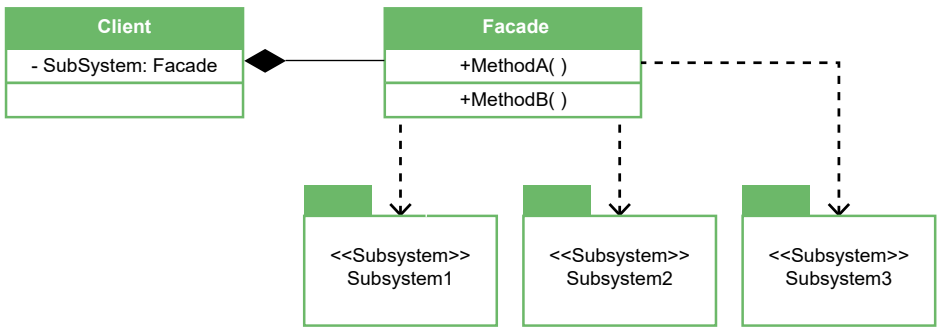


Figura 2.19. Patrón Fachada (Facade)

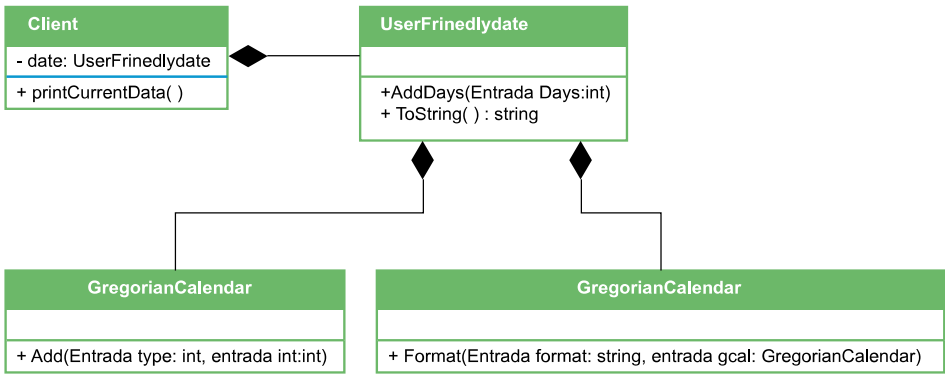


Figura 2.20. Ejemplo del patrón Fachada

2.2.6. Patrón peso ligero (FlyWeight).

Problema:

¿Cómo se puede aumentar el rendimiento o uso de memoria de una aplicación que posee elementos redundantes?

¿Cómo se pueden agrupar ciertas características de algunos elementos, de modo de asegurarnos que siempre sean iguales en todos los objetos?

Solución:

- Se define una clase Flyweight que contenga las características comunes.
- Luego todos los elementos redundantes hacen referencia a un mismo objeto de la clase Flyweight. Con lo anterior se reduce el uso de memoria y se elimina la redundancia de estas características. Las características referencian al mismo objeto y cambiar una implica cambiarla solo en un objeto.

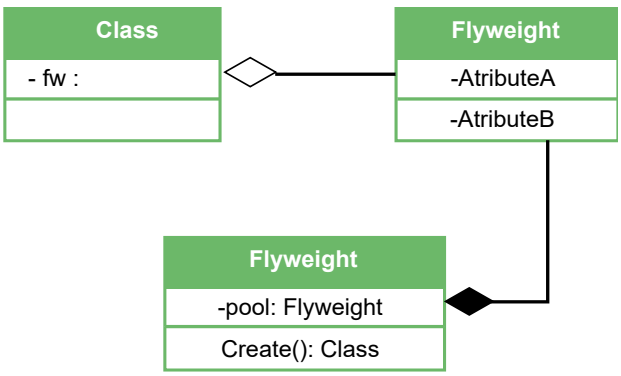


Figura 2.21. Patrón Peso Ligero (Flyweight)

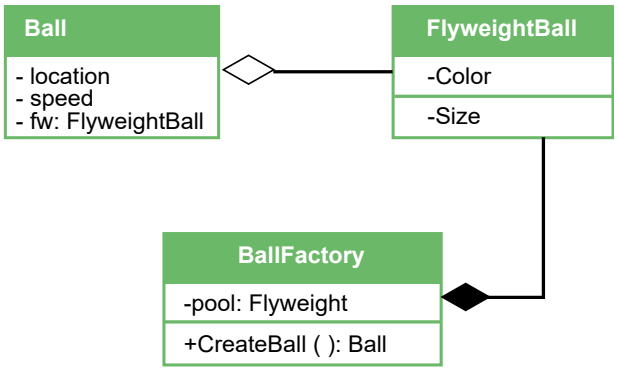


Figura 2.22. Ejemplo del patrón Peso Ligero

2.2.7. Patrón proxy.

Problema:

¿Qué hacer para controlar el uso de recursos de un objeto determinado dada su complejidad o uso de memoria?
¿Qué hacer para que los recursos se consuman solo si es necesario?

Solución:

- Se crea un intermediario con la misma interfaz que el objeto final. El intermediario contiene un objeto de la clase que se va a utilizar. Este objeto se crea solo cuando sea necesario.
- Este intermediario es el llamado proxy. Al compartir la interfaz con la clase final su uso es transparente para la clase cliente.

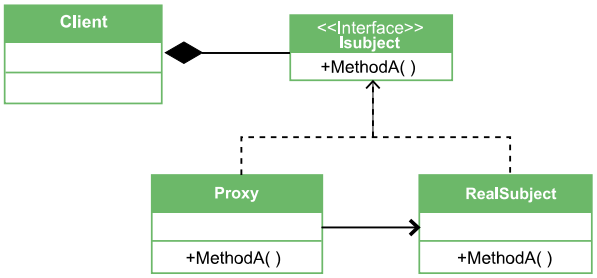


Figura 2.23. Patrón Proxy

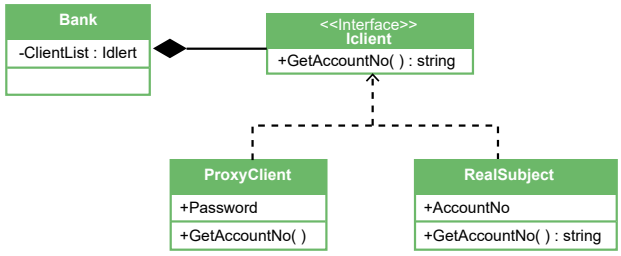


Figura 2.24. Ejemplo de patrón Proxy

2.3. Patrones de comportamiento.

Los patrones de este tipo son utilizados para organizar, manejar y combinar comportamientos.

Relación de patrones de comportamiento.

2.3.1. Cadena de Responsabilidades (Chain of Responsability): se usa para permitir que ante una petición se realicen acciones distintas dependiendo del estado del sistema.

2.3.2. Comando (Command): se usa para manejar acciones de forma independiente y generalizable, de modo que se pueden hacer invocaciones con estas acciones sin necesidad de saber quien la realiza realmente.

2.3.3. Intérprete (Interpreter): se usa para representar la gramática de un cierto lenguaje para poder interpretarlo.

2.3.4. Iterador (Iterator): se usa para recorrer objetos sin necesidad de conocer su estructura.

2.3.5. Mediador (Mediator): se usa para desacoplar la interacción entre distintos elementos, permitiendo que todos se comuniquen a través de un objeto común reduciendo así la complejidad de la interacción.

2.3.6. Recuerdo (Memento): se usa para manejar estados de la aplicación permitiendo volver a estados anteriores.

2.3.7. Observador (Observer): se usa para definir una dependencia de uno a muchos entre objetos de forma que muchos objetos puedan enterarse de los cambios en uno de ellos.

2.3.8. Estado (State): se usa para permitir que la ejecución de las acciones de una clase sea dependiente del estado en que se encuentra. El estado de un objeto puede variar en tiempo de ejecución.

2.3.9. Estrategia (Strategy): se usa cuando se quiere tener diversos algoritmos para realizar una tarea y poder cambiarlos en tiempo de ejecución.

2.3.1.0. Método Plantilla (Template Method): se usa para reutilizar el esqueleto de un algoritmo cambiando sólo el contenido de los pasos de éste último.

2.3.11. Visitante (Visitor): se usa para definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las cuales opera.

3. Patrón modelo vista controlador (MVC).

MVC son las siglas de Modelo Vista Controlador, que es un patrón de arquitectura de software cuya función es subdividir una aplicación en tres módulos o capas que corresponden a la vista del usuario (la interfaz a la que accede el usuario), una lógica de control para capturar los eventos que el usuario ha generado a través de la interfaz, y un modelo que gestiona los datos según le indique la lógica de control (GRAMMA,1994).

Este patrón tiene su origen en la forma como se hacían las interfaces en el lenguaje de programación Smalltalk-80 (GRAMMA,1994).

El objetivo del patrón MVC es desacoplar la presentación de la información (vista) de su representación (modelo) para así reducir la complejidad en el diseño arquitectónico de la interfaz de usuario e incrementar la flexibilidad y mantenibilidad del código (GRAMMA,1994).

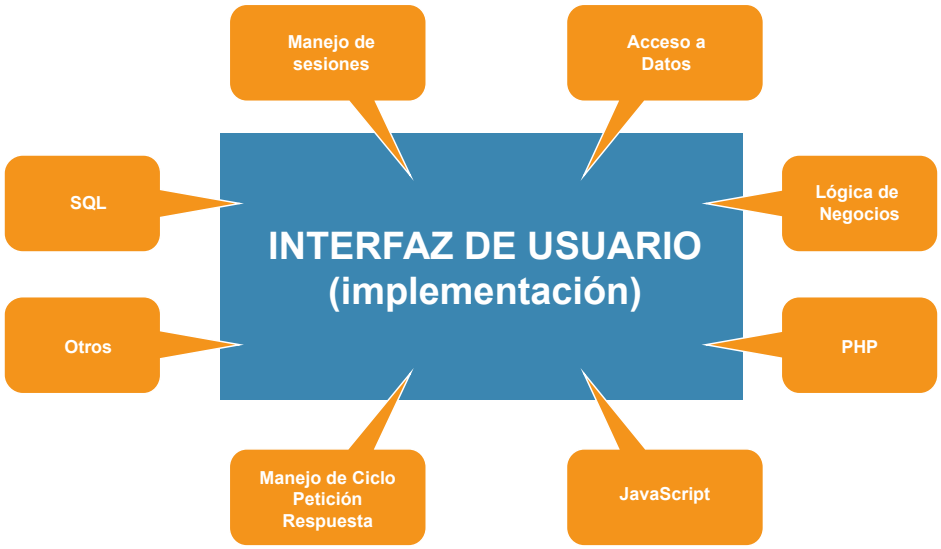


Figura 3.1. Sistema sin modelo MVC

Vista.

Este presenta el modelo en un formato adecuado para interactuar, usualmente la interfaz de usuario. Por lo tanto, la vista es la encargada de presentar los datos al usuario y la interfaz necesaria para modificarlos.

Controlador.

Este responde a eventos que son generalmente acciones del usuario e invoca cambios en el modelo y probablemente en la vista.

Desde este perspectiva el controlador sería la unidad central que comunica la vista con el modelo y viceversa asociando los eventos del usuario con los cambios que se producirán en el modelo.

También devuelve los datos resultantes que genere el modelo a la vista que corresponda.

Modelo.

Esta es la representación específica de la información con la cual el sistema opera y se compone por el Sistema de Gestión de Base de Datos (SGBD) y la lógica de negocio. La lógica de negocio asegura la integridad de estos y permite derivar nuevos datos.

El Sistema de Gestión de Base de Datos (SGBD) será el encargado de almacenar los cambios generados por la lógica de negocio. Ejemplos de SGBD son MySQL, Oracle, SQLServer, entre otros.

Es recomendable usar una capa de abstracción extra como las suministradas por las tecnologías Data Access Object (DAO), ODBC, JDBC, entre otras para el acceso al motor de bases de datos ya que permiten la separación completa entre el controlador y el motor de bases de datos.

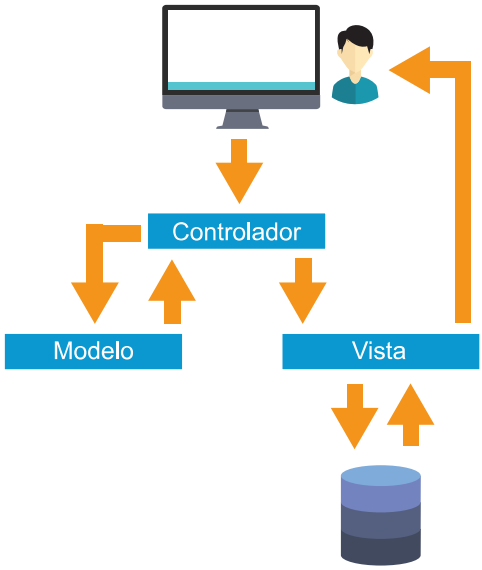


Figura 3.2. Sistema con el modelo MVC

Aunque se pueden encontrar diferentes implementaciones de MVC, el flujo de control generalmente es el siguiente:

a. El usuario interactúa con la interfaz de alguna manera (ej. presionando un botón, un enlace).
b. El controlador recibe por parte de los objetos de la interfaz vista la notificación de la acción solicitada por el usuario.
c. El controlador accede al modelo para consultar o actualizar los datos enviados por el usuario.
d. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario.
e. La vista usa el modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo.
f. En algunas implementaciones la vista no tiene acceso directo al modelo dejando que el controlador envíe los datos del modelo a la vista.
g. La interfaz espera por nuevas interacciones de usuario para iniciar nuevamente el ciclo.

4. Patrones de arquitectura para aplicaciones empresariales.

Además de los patrones introducidos por la banda de los cuatro (GoF) y el patrón MVC introducido en el lenguaje Smalltalk-80 existen otros aportes como los expuestos por Martin Fowler en su libro: Patrones de arquitectura para aplicaciones empresariales (Patterns of Enterprise Application Architecture) .

En su libro Fowler introduce una serie de patrones de común uso en el ámbito de aplicaciones empresariales especialmente las que acceden bases de datos y trabajan con usuarios de manera simultánea o concurrente.

En este recurso se introduce el patrón “registro activo” que puede ser usado de manera intensiva en la etapa de desarrollo del sistema de información. El aprendiz puede profundizar los demás patrones de Martin Fowler a través de la bibiografía que se especifica en este recurso.

4.1 patrón registro activo (active record).

Es un objeto que representa un registro de una tabla o vista de una base de datos. Además encapsula el acceso a la base de datos y agrega la lógica del dominio sobre esos datos (FOWLER,2002).

Este patrón se usa con mucha frecuencia en las aplicaciones que utilizan formularios o formas para capturar un registro de la base de datos que luego será modificado o para capturar un registro que será agregado a esa misma base de datos.

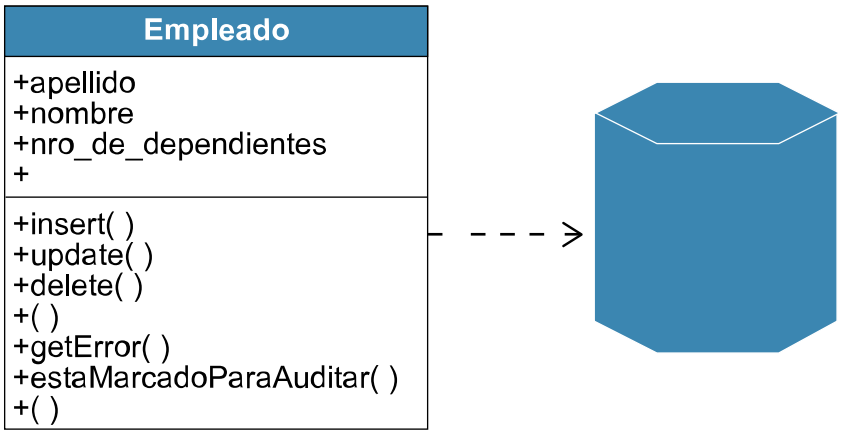


Figura 4.1. Patrón Registro Activo

La estructura de datos de un registro activo debe coincidir exactamente con la de la base de datos, es decir, un campo de la clase representar cada campo del registro en la base de datos.(FOWLER,2002)

El registro activo generalmente tiene métodos para hacer lo siguiente (FOWLER,2002):

a. Construir una instancia de la clase Registro Activo a partir de los resultados de una sentencia SQL.
b. Construir una instancia para luego insertar en la base de datos.
c. Métodos de búsqueda para representar sentencias SQL comúnmente usadas y retornar objetos del tipo Registro activo.
d. Actualizar la base de datos e insertar en esta los datos del Registro Activo.
e. Métodos para traer y definir los campos (setters and getters).
f. Métodos para implementar algunas reglas de negocio.

5. Cómo elegir un patrón de diseño.

Para seleccionar el patrón adecuado se pueden seguir los siguientes pasos:

1. Entienda el contexto donde quiere aplicar el patrón
2. Revise los patrones ya estudiados
3. Identifique y estudie los patrones que pueden aplicar
4. Aplique el patrón de la manera adecuada.

Al usar los patrones de diseño se debe evitar lo siguiente:

1. Usar un patrón equivocado.
2. Usar demasiados patrones que puedan complicar la implementación.

GLOSARIO

DAO: Data Access Object. Software intermedio que permite el acceso a una base de datos.

Encapsulamiento: propiedad de la programación orientada a objetos en la cual sólo se hacen visibles algunos atributos y métodos de estos últimos.

Herencia: cualidad de la programación orientada a objetos en la cual estos últimos tienen los atributos y métodos de una clase superior.

JDBC: Java Database Connectivity. Software intermedio que permite el acceso a bases de datos.

MVC: Model-View-Controller. Tipo de arquitectura que divide la interacción entre el usuario y el sistema en tres capas denominadas Modelo-Vista-Controlador.
ODBC: Open DataBase Connectivity : Software intermedio que permite el acceso a bases de datos.

Patrón de diseño: solución probada y documentada a un problema de diseño de software.

Polimorfismo: propiedad de la programación orientada a objetos que permite que un mismo objeto se comporte de manera diferente de acuerdo a los parámetros de entrada.

SGBD: sistema de gestión de bases de datos.

BIBLIOGRAFÍA

Gramma, E., Helm, R., Johnson, R., Vlissides, J., (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* USA: Addison-Wesley.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* USA: Addison-Wesley.

CONTROL DEL DOCUMENTO

CONSTRUCCIÓN
OBJETO DE
APRENDIZAJE

PATRONES DE DISEÑO	
Centro Industrial de Mantenimiento Integral - CIMI Regional Santander	
Líder línea de producción:	Santiago Lozada Garcés.
Asesores pedagógicos:	Rosa Elvia Quintero Guasca. Claudia Milena Hernández Naranjo.
Líder expertos temáticos: **Experto temático:** **Experto temático:**	Rita Rubiela Rincón Badillo. César Marino Cuellar. (V1) Nelson Mauricio Silva Maldonado.(V2)
Diseño multimedia:	Tirso Fernán Tabares Carreño.
Programador:	Francisco José Lizcano Reyes.
Producción de audio:	Víctor Hugo Tabares Carreño.