PATRONES DE DISEÑO DE SOFTWARE

RAFAEL VILLEGAS

FELIPE CORTES

INGENIERÍA DE SOFTWARE

INGENIERÍA DE SISTEMAS

UNIVERSIDAD EAFIT

MEDELLÍN 2018-2

TEMPLATE METHOD (PATRON DE METODO DE LA PLANTILLA)

El patrón de diseño Template Method forma parte de la familia de patrones denominados de comportamiento. Este tipo patrones ayudan a resolver problemas de interacción entre clases y objetos.

La solución que propone el patrón es abstraer todo el comportamiento que comparten las entidades en una clase (abstracta) de la que, posteriormente, extenderán dichas entidades. Esta superclase definirá un método que contendrá el esqueleto de ese algoritmo común (método plantilla o template method) y delegará determinada responsabilidad en las clases hijas, mediante uno o varios métodos abstractos que deberán implementar.

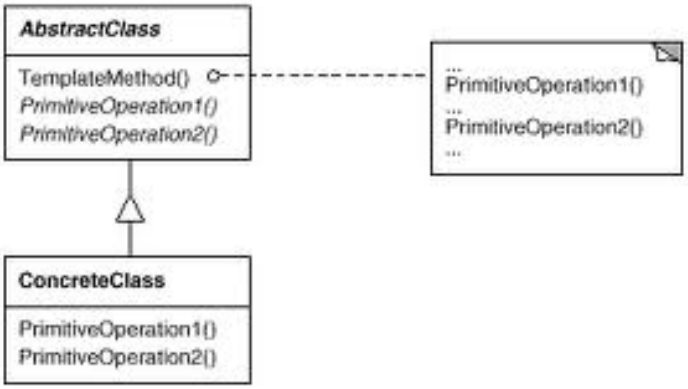
Motivación: Imagínese juegos de mesa como Monopoly, Risk, Parques, Escalera, entre otros. Todos esos juegos sin importar que se jueguen de formas muy diferentes, poseen ciertas características en común, que los permiten englobar en una super clase, que tendrá los métodos como mover pieza, tirar dado, retroceder pieza, ya que estos métodos están presentes en todos lo juegos sin importar cual sea. Pero como sabemos estos juegos son diferentes por ciertas propiedades que tengan, por lo cual son instancias de la super clase y estas tienen otros métodos propios que ya identifican a la clase en específico. Por lo que la plantilla seria la super clase de juego de mesa y las instancias o entidades, los juegos de mesa en particular.

Aplicabilidad: Este criterio se usa mucho cuando varias clases poseen el mismo comportamiento o tienen propiedades muy similares que pueden ser extraídas en una clase general. También se utiliza para que muchas clases con mismos métodos hereden de una clase mayor estos métodos y no estén repetidos en todas las clases hijas.

De los criterios para aplicar este patrón están:

- Se quiera factorizar el comportamiento común de varias subclases.
- Se necesite implementar las partes fijas de un algoritmo una sola vez y dejar que las subclases implementen las partes variables.
- Se busque controlar las ampliaciones de las subclases, convirtiendo en métodos plantillas aquellos métodos que pueden ser redefinidos.

Estructura:



Se puede observar como la clase plantilla es la clase Abstracta y de esta heredan clases concretas, las cuales heredan los métodos que contiene la clase abstracta y además contienen algunos métodos variables que tengan en si las hijas.

Participantes:

- **AbstractClass:** Es la clase plantilla, la cual contiene los métodos plantilla de todas las clases. Estos métodos son los métodos que están presentes en todas las clases hijas. Los métodos plantilla pueden rescribirse, sin embargo, son heredados por todas las instancias concretas de la clase plantilla. En general, extrae el comportamiento en común que contengan todas las clases.
- **ConcreteClass:** Es la clase instanciada de la plantilla, esta clase hereda todos los métodos plantilla de la clase plantilla, lo cual tiene todos los comportamientos básicos. Sin embargo, puede tener además otros métodos variables, que si son propios de dicha clase, ya que no los poseen otras clases que sean instanciadas de la misma clase plantilla.

Colaboraciones:

Solo hay presente un tipo de interrelación, la cual es herencia. Esta esta presente, ya que las clases concretas heredan los comportamientos básicos de la clase plantilla. Es necesario que sea herencia porque todas las clases hijas tienen que tener los mismos métodos por las cuales se creo la clase plantilla y heredarlas de la misma.

Consecuencias:

- Favorece la reutilización de código, se usa en la creación de bibliotecas
- Lleva a una estructura de control invertido (La superclase invoca los métodos de las subclases) – Principio de Hollywood
- Puede producir ambigüedad si es escrito de una manera inadecuada

PROXY

El patrón de diseño Proxy es un patrón de tipo estructural que pretende crear un intermediario de un objeto, para así controlar el acceso a este.

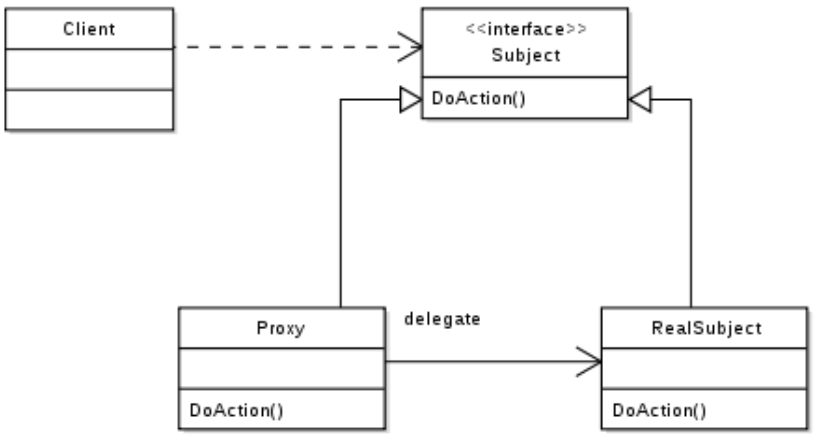
En su sentido más general, un proxy es una clase que funciona como una interfaz hacia otro objeto. Es un objeto que es llamado por el cliente para acceder al objeto que realmente se quiere acceder en forma de referencia. De esta forma, el cliente nunca hace contacto directo con el objeto real que pretende usar, ya que todo este tipo de interacciones pasan a través del proxy.

Motivación: Suponga a un editor que puede incluir objetos gráficos dentro de un documento. Este necesita que la apertura del documento sea rápida, pero se sabe que la creación de algunos objetos, en especial los gráficos, es muy costosa. Por ello, no es necesario cargar todos los objetos del documento justo cuando se abre, sino que es mejor retrasar el costo de cargar uno de estos objetos hasta que realmente sean necesarios. Para ello se debe crear el proxy cuando se carga el documento y que el este llame a cada imagen cuando esta sea demandada.

Aplicabilidad: Un objeto proxy se usa cuando se necesita una referencia más elaborada y flexible hacia un objeto que un simple apuntador, o cuando se quiere simplificar el control de un objeto. Esta puede ser de diferentes tipos según lo que se pretende hacer:

- **Remoto:** es un representante local de un objeto remoto.
- **Virtual:** Crea objetos costosos bajo demanda
- **Protección:** Controla acceso a objeto original.
- **Referencia Inteligente:** Sustituto de puntero que permite hacer operaciones adicionales sobre el objeto.

Estructura:



Hay una interfaz que implementan todos los objetos que el proxy desea referenciar, además del mismo proxy. De esta forma el cliente puede usar los objetos y el proxy indiferentemente.

La Clase proxy mantiene una referencia al objeto real y proporciona una interfaz idéntica a la interfaz real que permite emular.

El objeto proxy redirige las peticiones del cliente al objeto real de acuerdo a ciertas condiciones.

Participantes:

- **Client:** hace las peticiones y los llamados para interactuar con objetos, usando una interfaz.
- **Subject:** interfaz que encapsula todo lo relacionado con los objetos específicos a usar. Establece el comportamiento que tendrán los objetos y como el cliente podrá interactuar con ellos.
- **RealSubject:** clase que representa los objetos reales del programa. Hereda de la interfaz Subject.
- **Proxy:** clase que representa los objetos proxy que pueden ser creados para recibir las instrucciones del cliente y delegarselas a los objetos reales siguiendo ciertos parámetros. Hereda de la interfaz Subject para tener el mismo comportamiento que los objetos reales.

Colaboraciones: La colaboración principal es la de delegación de actividades de parte del proxy hacia el objeto real. También hay que recalcar que hay herencia desde el proxy y el objeto real con respecto a la interfaz definida para interactuar con los objetos.

Consecuencias:

- **Remoto:** Oculta el hecho de que un objeto reside en otro espacio de direcciones.
- **Virtual:** puede realizar optimizaciones, como creación de objetos bajo demanda.
- **Protección y referencias inteligentes:** permiten realizar diversas tareas además de acceder al objeto.
- **Optimización COW (copy-on-read):** Retrasa la copia de un objeto hasta que sea necesario cambiarlo, ya que trabaja con referencias mientras que no se requiera cambiar el objeto.

Referencias

https://es.wikipedia.org/wiki/Patr%C3%B3n_de_m%C3%A9todo_de_la_plantilla

https://www.adictosaltrabajo.com/2011/10/04/patron-template-method/

http://migranitodejava.blogspot.com/2011/06/template-method.html

https://www.ecured.cu/Template_Method

https://es.wikipedia.org/wiki/Proxy_(patrón_de_diseño)

https://en.wikipedia.org/wiki/Proxy_pattern