Definiciones:

Ingeniería del software: aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo operación y mantenimiento de software.

Modelo: Una representación o especificación desde un determinado punto de vista y con un objetivo concreto.

Diseño: Conjunto de planes y decisiones para definir un producto con los suficientes detalles como para permitir su realización física de acuerdo a unos requisitos.

Patrón de diseño: solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en el desarrollo de software.

Diferencias y semejanas: Prototype & Abstract Factory

Ambos patrones se centran en la creación de objetos, sin embargo, hasta ahí llega su semejanza puesto que el patrón de diseño prototipo nos permite generar clases dinámicamente en tiempo de ejecución, sin embargo la factoría abstracta aísla clases concretas para el cliente. Además, la factoría abstracta relaciona familias de clases que dependen entre ellas, mientras que el prototipo nos sirve para especificar en un momento dado el tipo de una clase abstracta a la hora de hacer uso de ella.

*Refactoring* Define.

El refactor es una técnica en el proceso de desarrollo software que nos permite la reestructuración del código fuente de nuestro proyecto alterando la estructura interna, pero no su comportamiento. (No sé el proceso de refactor so…)

Examen en B (en si para los que no sepan musica):

Ejercicio 1:

Porqué añadir la clase banco?

En una de las restricciones, el enunciado nos dice que “las cuentas pueden tener varios propietarios y han de ser del mismo banco”. Para poder controlar dicha restricción necesitaremos modelar la clase Banco y añadir una relación entre cuenta y banco, que obligue a la cuenta a estar relacionada con un único Banco obligatoriamente. Además, para modelar dicha restricción deberemos también relacionar los clientes con todos los bancos en los que es cliente, y para estar en el modelo, debe estar como mínimo en 1 banco.

Puesto que nuestro banco necesitará también las cuentas de débito, hemos pensado en realizar una clase abstracta llamada Cuenta, de la que heredarán 2 tipos concretos de cuenta, Corriente y de Débito. La cuenta corriente no añade ningún comportamiento adicional a la clase padre, sin embargo, la cuenta de Débito añade un nuevo atributo y un nuevo constructor, además de redefinir el método de reintegro.

En el enunciado del problema se nos especifica que no se podrá realizar una operación de reintegro cuando esté en rojo la cuenta, sin embargo, no se nos comenta nada acerca de poder realizar una operación de reintegro estando la cuenta en negro, pero que la deje en rojo. Supondremos que este comportamiento puede darse y que se puede retirar más dinero que saldo haya en la cuenta siempre que esté la cuenta en Negro.

Como se nos indica en el enunciado, el cliente deberá tener una cuenta que sea la suya principal, por tanto creamos una relación, que es la manera en UML de tener una “referencia” entre dos clases.

En cuanto a las operaciones, vamos a necesitar añadir una operación que nos permita transitar entre estados. Dicha operación será llamada al final de cada operación para comprobar si ha habido algún cambio entre estados. La operación se llama *checkSaldo()*. Otro motivo por el cual añadir dicha operación es que simplifica las transiciones entre estados, puesto que no tienes que añadir las mismas transiciones para todas las operaciones. Otra opción a parte de crear el *checkSaldo()* es obligar a que todas las operaciones devuelvan el saldo después de la operación, y así llamar a la operación *saldo()* al final de cada llamada a una operación.

(Vamos a utilizar la última opción, por algún motivo me resulta más atractiva).

En cuanto a la creación de la clase Cuenta, el Banco se encargará de crearlos llamando a las operaciones correspondientes.

Además, se nos dice que las cuentas pueden tener más de un autorizado y más de un Propietario, por tanto, añadiremos 2 operaciones en la clase banco que nos permita añadir dicha información a una cuenta específica.

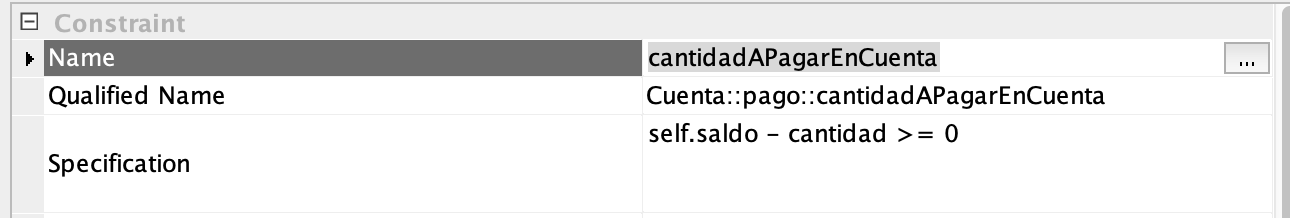
Por último, las operaciones de ingreso, pago, etc, tendrán que estar en la clase Cliente, así sabremos si el usuario puede realizar dicha operación o no.

En cuanto a las restricciones se han añadido pre y post-condiciones a las operaciones para definir en alto nivel su comportamiento.

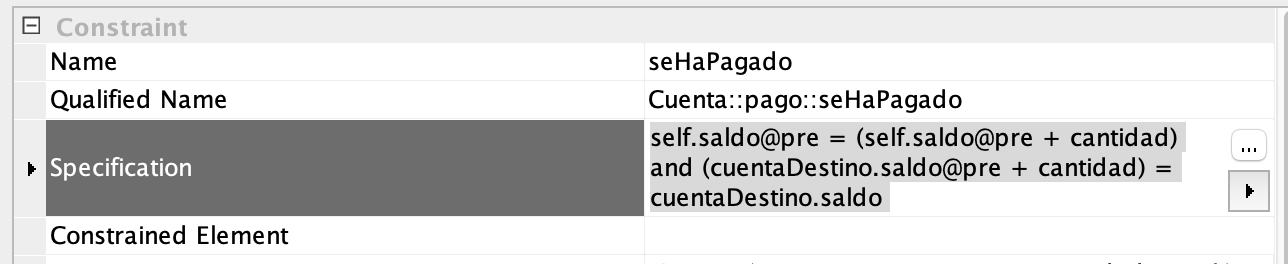
Se han añadido las siguientes pre y post-condiciones:

Cuenta::pago()

Pre



Post

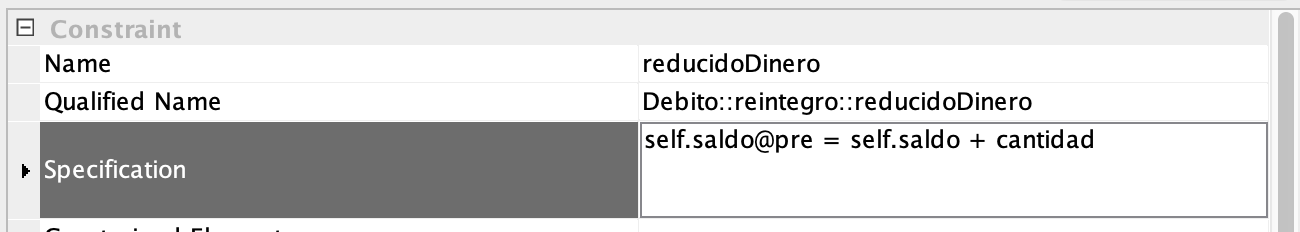


Debito::reintegro()

Pre

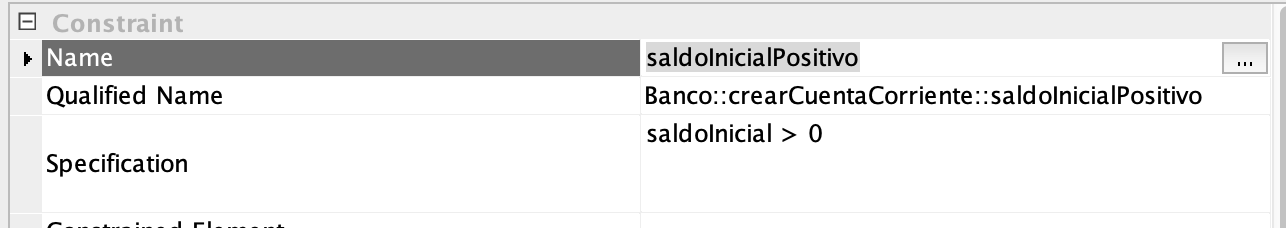


Post

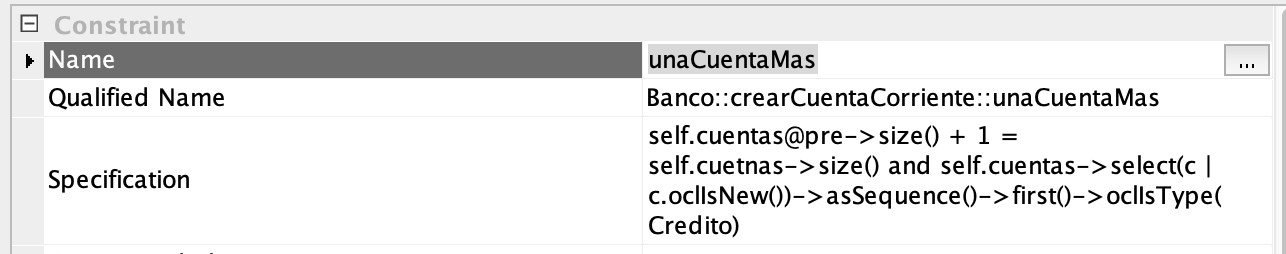


Banco::crearCuentaCorriente()

Pre

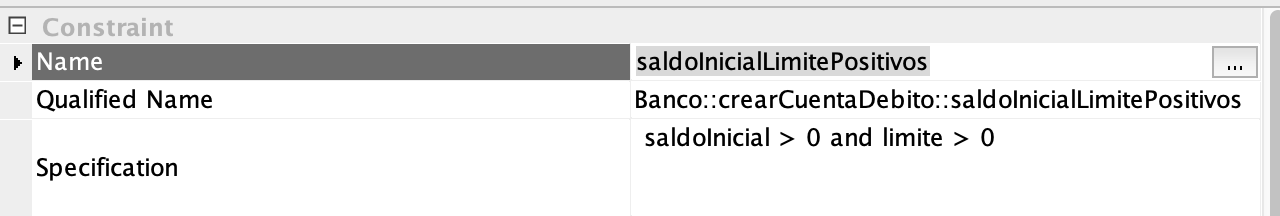


Post



Banco::crearCuentaDebito()

Pre

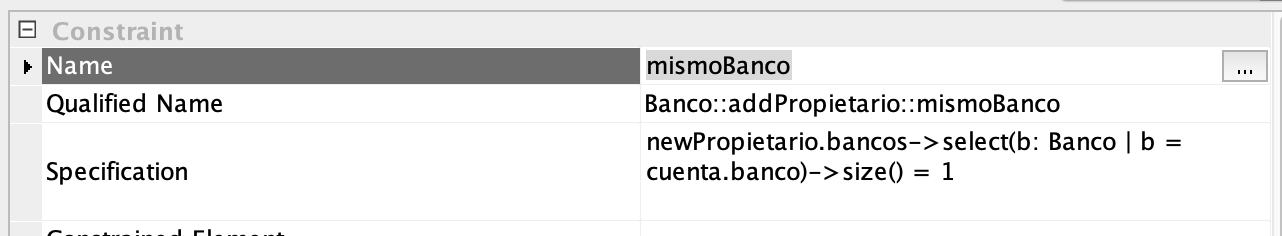


Post

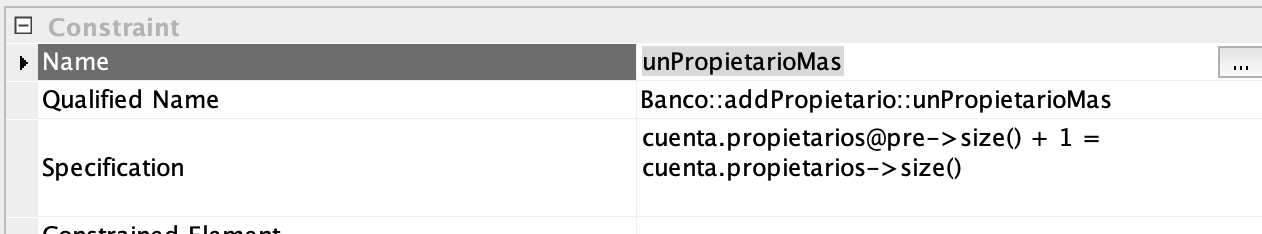
Misma post-condición que creacUentaCorriente:::unaCuentaMas

Banco::addPropietario

Pre



Post



Banco::addAutorizado

Mismas pre y post-condiciones que en Banco::addPropietario, pero cambiando en la post-condición cuenta.propietarios por cuenta.autorizados.

Hay unas cuantas pre y post más de la clase Cuenta pero hace referencia a las operaciones que deberían estar en Cliente, y como no lo he cambiado, lo cambiaré luego si me apetece

Diagrama UML:

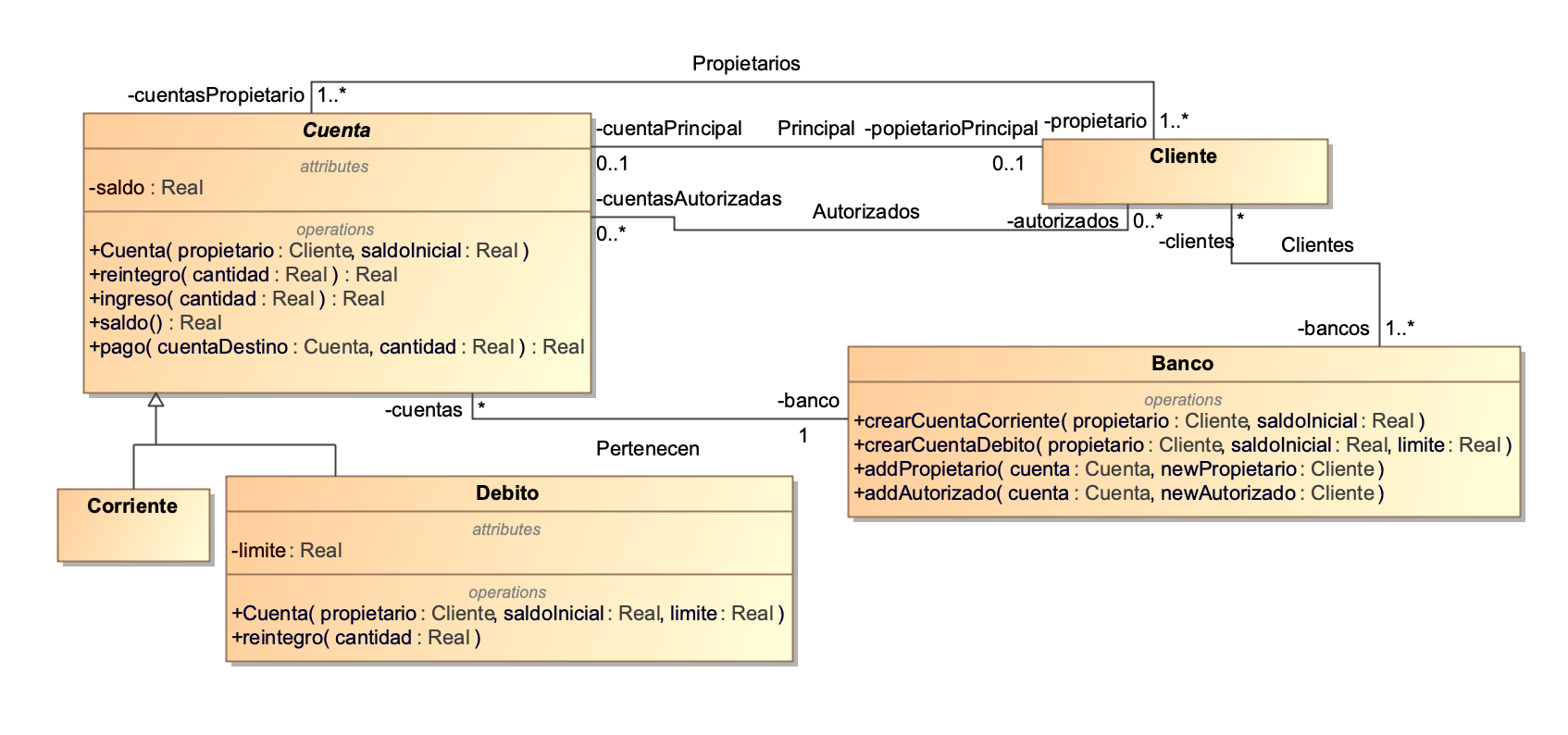
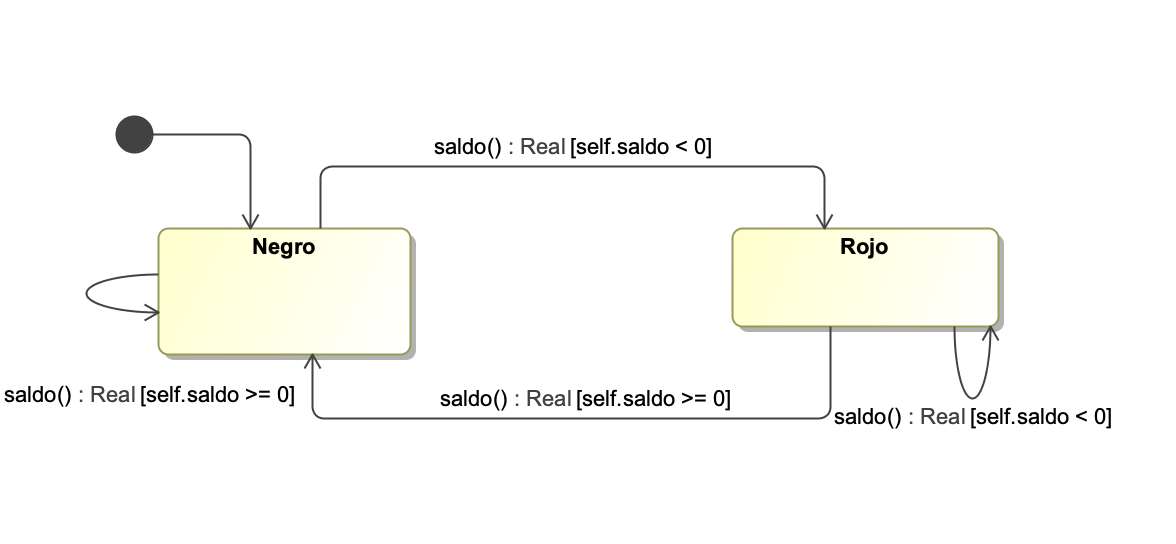


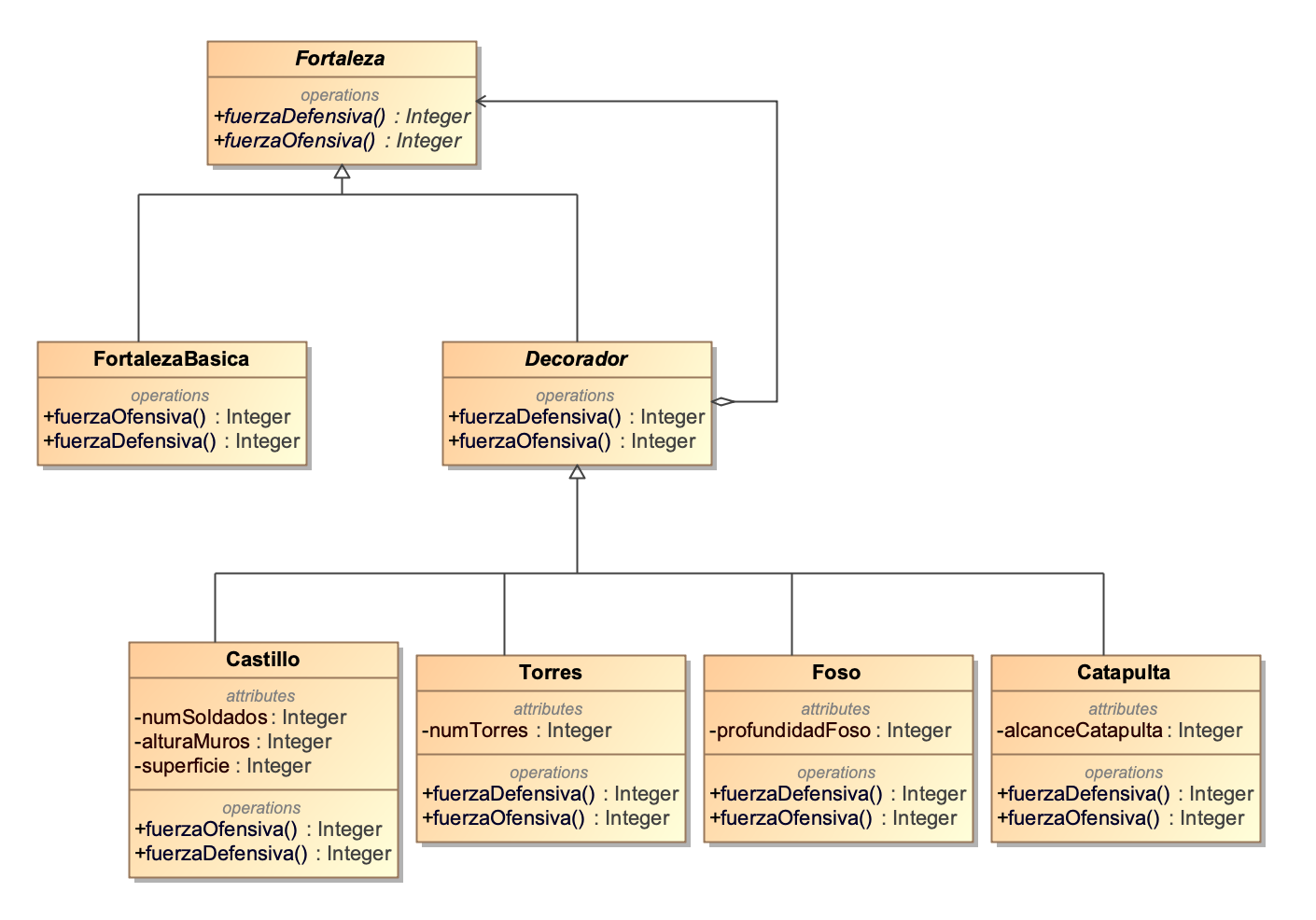
Diagrama de estados:  


Ejercicio 2:

1. Puesto que Java no permite herencia múltiple y en la solución observamos como las clases van heredando de 2 y 3 clases para crear todas las combinaciones posibles de fortalezas. Para poder realizar esta solución en Java podemos utilizar varios enfoques. Uno de ellos es utilizar una interfaz por cada clase de la que se hereda. Por ejemplo, crear una interfaz *ICastillo,* que hereda de *Fortaleza*, luego una interfaz *ITorre*, que también herede de *Fortaleza,* que las clases *Castillo* y *Fortaleza* implementen sus respectivas interfaces. Luego nuestra clase *CastilloTorre* que tenga una referencia a *Castillo* otra a *Torre* y que implemente sendas Interfaces. Otra posible solución es utilizar los roles. Ambas soluciones son muy complicadas, ensucian el código y crean cantidad de clases innecesarias para poder implementar. Imaginemos que queremos añadir una nueva defensa a la Fortaleza, llamada *Cañon*. A la hora de añadir el cañón tendríamos que modificar cantidad de clases para poder tener todas las combinaciones posibles de fortalezas, haciendo que a la hora de ampliar el código sea muy complicado de modificar y mantener.
2. Para poder resolver este problema utilizaremos el patrón decorador. Para ello utilizaremos la clase Abstracta *Fortaleza* como Componente al que vamos a decorar. Para ello heredarán 2 clases, una que será el componente básico, en nuestro caso una *FortalezaBásica* y nuestro *Decorador*, que mantendrá una referencia a la Fortaleza. De nuestro *Decorador* heredarán todas las clases que heredan de *Fortaleza* en el modelo original, que son, *Castillo, Foso, Catapulta* y *Torre*. La clase *Fortaleza* tendrá sus dos métodos abstractos *fuerzaDefensiva()* y *fuerzaOfensiva()*, que tendrán también *FortalezaBásica* y *Decorador*. Estos métodos también los tendrán nuestras clases que heredan de *Decorador*. Nuestro decorador a la hora de llamar a las funciones delegará su comportamiento a la referencia de nuestra Fortaleza, de este modo, podremos extender el comportamiento de la Fortaleza mediante asignación dinámica. Este patrón resuelve por completo el problema, aunque tengamos que tener bastantes pequeños objetos, pero sin duda reduce la cantidad de clases a crear respecto a la solución antigua.

Para el esquema del código Java vamos a suponer que la fuerza defensiva y ofensiva son los valores de los atributos de cada clase en concreto. Por ejemplo, la fuerza defensiva del foso es el valor de su profundidad, mientras que su valor de ataque será 0.

Diagrama UML de la solución propuesta:



Código Java:

Clase Fortaleza:

*package* dec2017;

*public abstract class* Fortaleza {

*abstract int* fuerzaDefensiva();

*abstract int* fuerzaOfensiva();

}

Clase FortalezaBasica:

*package* dec2017;

*public class* FortalezaBasica *extends* Fortaleza {

@Override

*int* fuerzaDefensiva() {

*return* 0;

}

@Override

*int* fuerzaOfensiva() {

*return* 0;

}

}

Clase Decorador:

*package* dec2017;

*public abstract class* Decorador *extends* Fortaleza{

*private* Fortaleza fortaleza;

*public* Decorador(Fortaleza fortaleza) {

*this*.fortaleza = fortaleza;

}

@Override

*int* fuerzaDefensiva() {

*return* fortaleza.fuerzaDefensiva();

}

@Override

*int* fuerzaOfensiva() {

*return* fortaleza.fuerzaOfensiva();

}

}

Clase Castillo:

*package* dec2017;

*public class* Castillo *extends* Decorador {

*private int* soldados;

*private int* alturaMuros;

*private int* superficie;

*public* Castillo(Fortaleza fortaleza, *int* soldados, *int* alturaMuros, *int* superficie) {

*super*(fortaleza);

*this*.soldados = soldados;

*this*.alturaMuros = alturaMuros;

*this*.superficie = superficie;

}

@Override

*int* fuerzaOfensiva() {

*return super*.fuerzaOfensiva() + soldados;

}

@Override

*int* fuerzaDefensiva() {

*return super*.fuerzaDefensiva() + alturaMuros + superficie;

}

}

Clase Catapulta:

*package* dec2017;

*public class* Catapulta *extends* Decorador {

*private int* alcance;

*public* Catapulta(Fortaleza fortaleza, *int* alcance) {

*super*(fortaleza);

*this*.alcance = alcance;

}

@Override

*int* fuerzaDefensiva() {

*return super*.fuerzaDefensiva();

}

@Override

*int* fuerzaOfensiva() {

*return super*.fuerzaOfensiva() + alcance;

}

}

Clase Foso:

*package* dec2017;

*public class* Foso *extends* Decorador {

*private int* profundidad;

*public* Foso(Fortaleza fortaleza, *int* profundidad) {

*super*(fortaleza);

*this*.profundidad = profundidad;

}

@Override

*int* fuerzaDefensiva() {

*return super*.fuerzaDefensiva() + profundidad;

}

@Override

*int* fuerzaOfensiva() {

*return super*.fuerzaOfensiva();

}

}

Clase Torres:

*package* dec2017;

*public class* Torres *extends* Decorador {

*private int* altura;

*public* Torres(Fortaleza fortaleza, *int* alutra) {

*super*(fortaleza);

*this*.altura = alutra;

}

@Override

*int* fuerzaDefensiva() {

*return super*.fuerzaDefensiva() + altura;

}

@Override

*int* fuerzaOfensiva() {

*return super*.fuerzaOfensiva();

}

}

Main para los lectores del fichero:

*package* dec2017;

*public class* Main {

*public static void* main(String[] args) {

Fortaleza fortaleza = *new* Castillo(

*new* Foso(

*new* Catapulta(

*new* Torres(*new* FortalezaBasica(), 5)

, 5)

, 5)

, 5,5,5);

System.out.println(fortaleza.fuerzaDefensiva());

System.out.println(fortaleza.fuerzaOfensiva());

}

}