Instituto Tecnológico de Costa Rica

Escuela de Computación

Ingeniería en computación

Diseño de Software

Grupo 1

Profesor: Luis Chavarría

Sistema de Amortización

Josué Arrieta Salas, 2014008153

Fernando Molina

Cartago

Lunes 30 de mayo

Índice

Contenido

[SOLID 3](#_Toc449260759)

[SRP – Single Responsability Principle 3](#_Toc449260760)

[OCP – Open/Close Principle 4](#_Toc449260761)

[LSP – Liskov Substitution Principle 5](#_Toc449260762)

[ISP – Interface Segragation Principle 7](#_Toc449260763)

[DIP – Dependency Inversion Principle 8](#_Toc449260764)

[Patrones de diseño 10](#_Toc449260765)

[MVC 10](#_Toc449260766)

[Strategy 11](#_Toc449260767)

[DTO 13](#_Toc449260768)

[Factory (creacional) 14](#_Toc449260769)

[Ejecución de funcionalidades 16](#_Toc449260770)

[Caso 1 : Funcionalidad de inversión tradicional 16](#_Toc449260771)

[Modo GUI 16](#_Toc449260772)

[Modo Consola 17](#_Toc449260773)

[Caso 2 : Funcionalidad de inversión pactada 18](#_Toc449260774)

[Modo GUI 18](#_Toc449260775)

[Modo Consola 19](#_Toc449260776)

[Caso 3 : Funcionalidad de inversión certificada 20](#_Toc449260777)

[Modo GUI 20](#_Toc449260778)

[Modo Consola 21](#_Toc449260779)

[Bitácoras 22](#_Toc449260780)

[XML 22](#_Toc449260781)

[CSV 23](#_Toc449260782)

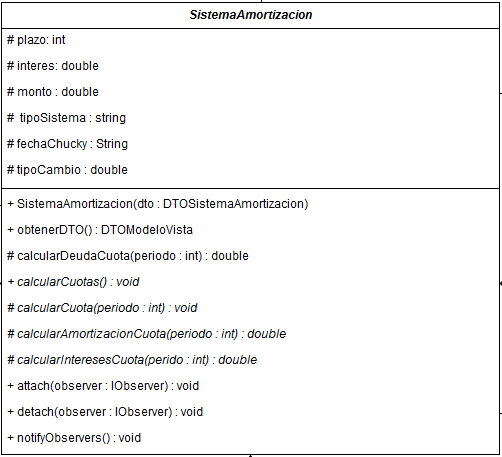
# SOLID

Para cada letra se debe:

* Screenshots con partes del diseño UML donde se cumple el principio.
* Segmentos de código donde se cumple el principio.
* Explicación o estrategias tomadas en consideración para el cumplimiento del principio.

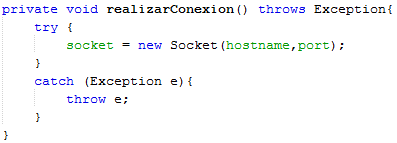
## SRP – Single Responsability Principle

Una entidad solo tiene una responsabilidad. Se deben agrupar responsabilidades similares. En el siguiente diseño se tiene la clase: SistemaArmotizacion.



SRP se ve cumplido a nivel de clase ya que esta clase solo tiene una responsabilidad: la lógica de negocios de un sistema de amortización. Solo posee atributos relacionados con esta lógica y realiza cálculos para un sistema de armotización. Solo posee una razón para cambiar ya que no mezcla responsabilidades.

A nivel de código se tiene el siguiente método:



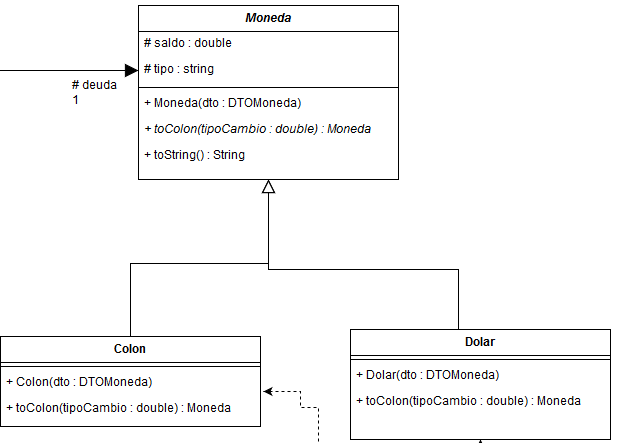
En este método SRP se ve cumplido ya que sólo posee 1 responsabilidad: realizar la conexión por medio de un socket a un back-end local. Solo posee una razón para cambiar.

Para cumplir SRP se trataron de seguir los siguientes puntos como estrategias:

* Promover siempre la alta cohesión para agrupar responsabilidades similares.
* Evitar por completo una clase Utilitaria enorme con un grado de acoplamiento altísimo.
* Asegurarse que la clase solo tiene una razón para existir, así para cambiar.
* Si un método o una clase se está haciendo excesivamente, es probable que tenga más de una responsabilidad (aunque no siempre).

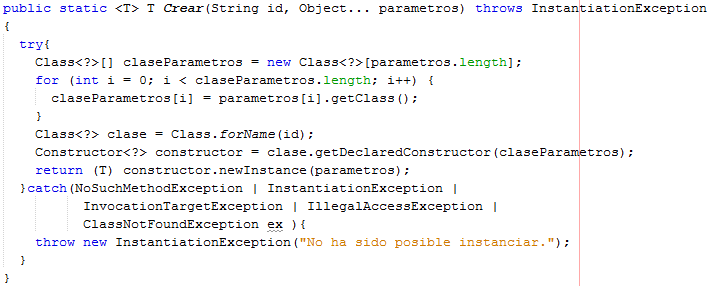
## OCP – Open/Close Principle

Las entidades deben ser abiertas para extensión y cerradas para modificación. Básicamente OCP se puede definir como: nunca modificar código existente y funcional. A nivel de diseño se tiene el siguiente ejemplo de OCP:



Digamos que se quiere extender la funcionalidad y crear un nuevo tipo de moneda, además de los ya existentes: Colon y Dolar. No es necesario cambiar el código en ninguna parte. Si no que está abierto para extensión: se agrega una nueva clase que herede de *Moneda* y se agrega el constructor particular de la nueva clase agregada.

A nivel de código de código se tiene el siguiente ejemplo donde se cumple OCP (es el método de creación de las fábricas):



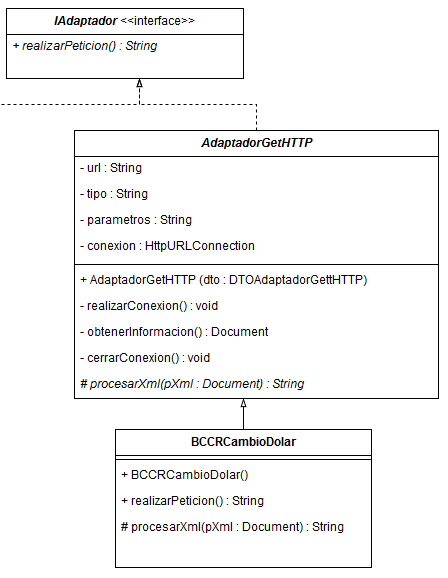
En el patrón de diseño Factory debe recibir un string o tipo, para indicar qué tipo de objeto se debe instanciar. Muchas veces se utiliza una estructura switch o if – else para la creación de objetos. Sin embargo este tipo de estructuras, si se necesita agregar la creación de un nuevo objeto, es necesario agregar un nuevo if o un nuevo case del switch. Esto viola OCP ya que se modifica código existente y dificulta el mantenimiento. Sin embargo si se utiliza la reflexión (como es este caso) tales estructuras if y switch no son necesarias, y en un futuro el código queda totalmente abierto para extensión y OCP se cumple.

Para cumplir OCP se siguieron los siguientes principios como estrategia:

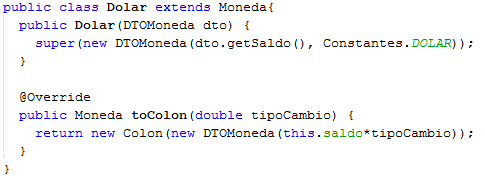
* No usar del todo estructura switch.
* No usar más de 2 if´s seguidos.
* Recurrir a la reflexión y polimorfismo.
* Agregar nuevas clases a las jerarquías para agregar nuevas funcionalidades.
* Sobrecargar sobrescribir métodos como medida para que el código este abierto a la extensión.

## LSP – Liskov Substitution Principle

Subclases deben poder ser tratadas como sus superclases. Estas deben poseer exactamente la misma interfaz. A nivel de diseño se tiene el siguiente ejemplo donde se cumple LSP (la jerarquía de adaptador para protocolo de transporte http, donde TODAS las sublcases se pueden tratar sin problema como su superclase: *AdaptadorGETHTTP*; ya que tienen la misma interfaz (conjunto de métodos públicos)):



Para el ejemplo de código se tomará la clase dólar que herada de moneda:



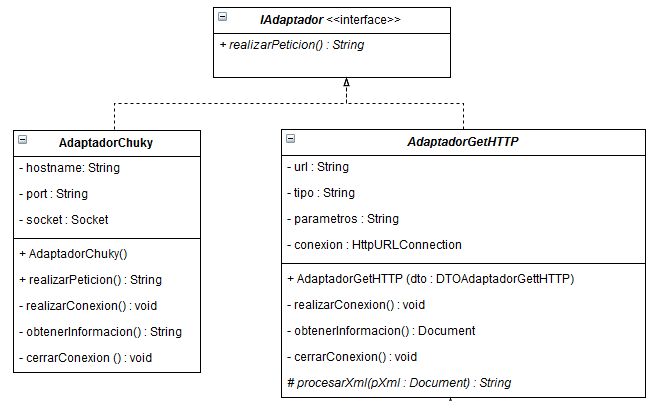
El código cumple Liskov ya que la clase Dólar en su interfaz no añade ningún método público nuevo y por lo tanto Dólar podrá ser tratada como su clase padre Moneda y Liskov se cumple.

Para cumplir LSP se siguieron los siguientes puntos como estrategia:

* En una jerarquía de clases asegurarse que la superclase defina los métodos públicos de las subclases.
* Nunca definir un método públic para una subclase ya que las superclase no lo posee y se podrá violar Liskov si esto se hace.
* Si hay que usar estructuras switch o if else para identificar el tipo de datos con el que se esté trabajando es posible que se esté violando LSP.
* Nunca debe existir el caso en que una subclase es tratada diferente como las demás subclases en una jerarquía.

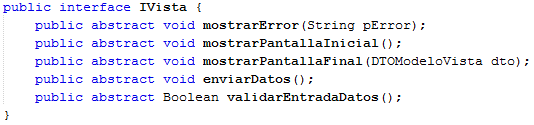
## ISP – Interface Segragation Principle

Una interfaz grande de propósito general no sirve, es mejor tener múltiples interfaces especializadas. Las interfaces tienen que ser lo más pequeñas posibles. Se tiene el siguiente diseño donde se cumple ISP:



La interface al solo poseer la firma de un método (la interface corresponde al protocolo de comportamiento para realizar peticiones específicas para el cumplimiento del patrón adaptador) cumple ISP automáticamente. Es una interfaz altamente especializada.

A nivel de código se tiene el siguiente ejemplo:



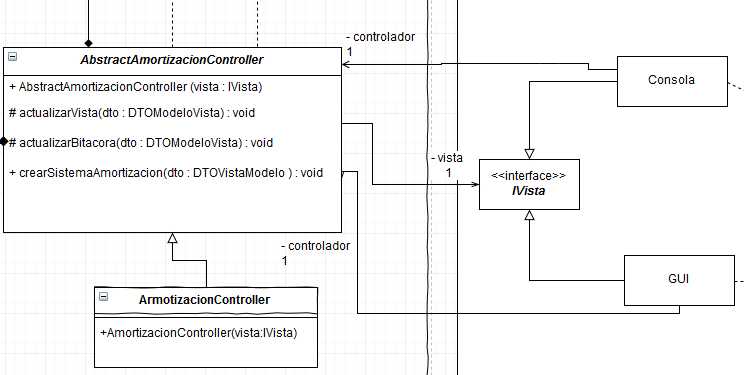
Sería la interfaz usada para el protocolo de comportamiento para las vistas. Es altamente especializada ya que solo posee firmas de métodos para la interacción con los usuarios (y validación de datos de entrada del usuario).

Para cumplir ISP se utilizaron los siguientes puntos como estrategia:

* Buscar interfaces pequeñas especializadas.
* Si hay una interfaz grande, es posible que ISP se esté violando.
* Para deshacer una interfaz grande, se puede aplicar herencia de interfaces.
* Firmas de métodos de interfaces deben estar relacionados y deben ir juntos en la misma de interfaz.

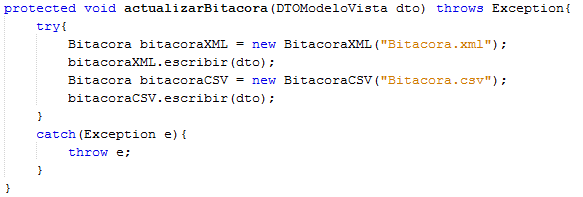
## DIP – Dependency Inversion Principle

Depender en abstracciones y no en objetos concretos. Se debe “desconectar” clases abstractas por medio de una capa de abstracción. Se tiene el siguiente diseño donde se cumple DIP:



Se usó DIP entre capas. Las interfaces no tienen referencia a un controlador concreto (y lo mismo que el controlador no tiene referencia a vistas concretas). Se depende en abstracciones (ya sea una clase abstracta o interfaz). En este caso 2 clases concretas no están relacionadas.

A nivel de código se tiene el siguiente ejemplo (dentro de la clase *AbstractArmotizacionController*):



El controlador no tiene referencia a una clase concreta, si no que a una clase *Bitacora* abstracta, de tal manera que DIP se ve cumplido. No se relacionan 2 clases concretas.

Para cumplir este principio se trataron de seguir los siguientes puntos como estrategia:

* Si se ven 2 clases concretas relacionadas, agregar una abstracción (ya sea una clase abstracta o una interface) entre ellas.
* Si se tiene una clase concreta, siempre es bueno realizar una jerarquía; para crear una superclase abstracta que funcione como abstracción.
* Ninguna variable debe tener referencia a una clase concreta, siempre a una abstracción.

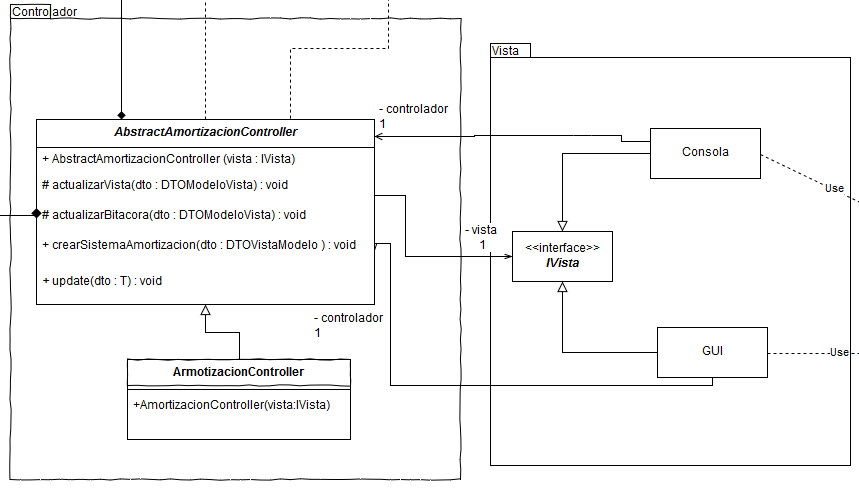
# Patrones de diseño

Para cada patrón se debe:

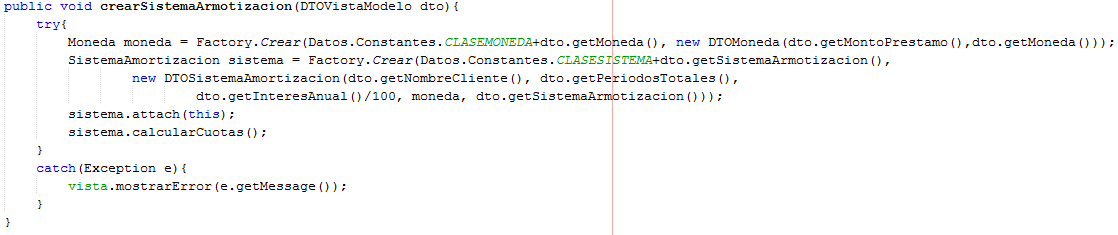
* Screenshots con partes del diseño UML donde se cumple el patrón.
* Segmentos de código donde se cumple el patrón.
* Explicación o estrategias tomadas en consideración para el cumplimiento del patrón.

## MVC

El patrón supone la separación de la lógica de negocios (modelo) y la interfaz; y que un controlador “amarre” dichas capas. Se tiene el siguiente diseño donde se implementa MVC:



Se puede que hay comunicación entre Las Vistas y Los Controladores. Las vistas interactuarán con el usuario y enviarán sus peticiones al controlador. Posteriormente el controlador determina la naturaleza de estas peticiones e interactuará con el modelo. El modelo es completamente agnóstico al controlador y a la interfaz. Hay una clara distinción de capas.

 A nivel de código se tiene el siguiente ejemplo:

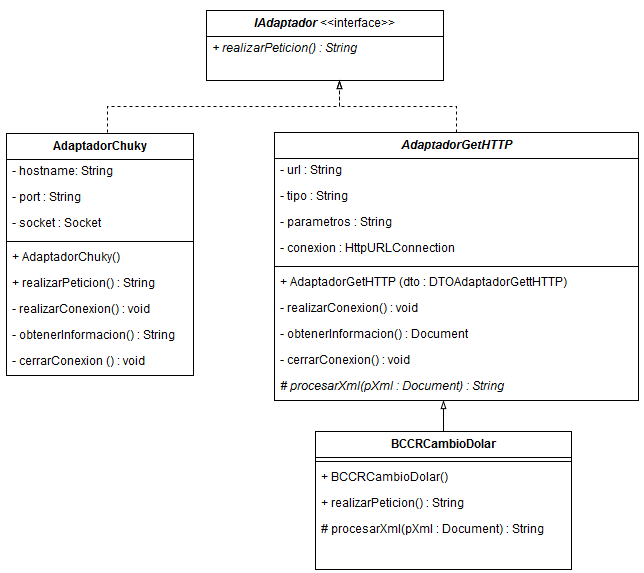
Tal ejemplo es código presenta en la clase *AbstractAmortizacionController*. Se puede ver cómo el controlador utiliza las clases del modelo para responder a una necesidad del negocio y a su vez determina la naturaleza de la solicitud de la interfaz.

Para cumplir MVC como estrategia se tomaron los siguientes puntos:

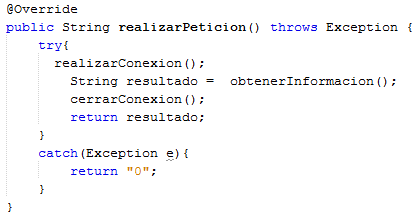
* La capa de vista solo debe poseer la lógica de presentación.
* La vista debe enviar solicitudes al controlador.
* El controlador siempre debe determinar la naturaleza de la solicitud.
* El controlador siempre debe retornar la respuesta a la vista.
* El modelo solo debe implementar la lógica de negocios.
* La capa del modelo debe ser completamente agnóstica a las demás.

## Adaptador (estructural)

El patrón de diseño adaptador se tiene cuando se tiene una clase cliente y otra proveedora, que sin embargo no son compatibles debido a protocolos de datos o transporte diferente. Es una especie de “traductor”. Se tiene el siguiente ejemplo a nivel de diseño:



0000 La interface *IAdaptador* establece el protocolo de comportamiento del adaptador. Posee un solo método para realizar peticiones específicas. Luego se pueden agregar otras clases con diferentes protocolos: localhost (AdaptadorChuky) o peticiones http(s) con protocolo de datos XML (*AdaptadorGetHTTP*). Es importante que al realizar la petición, devuelvan los datos como String, por lo tanto cada clase que implemente la interfaz debe manejar los datos y devolverlos en formato String.

 A Nivel de código se tiene el siguiente ejemplo en donde se implemente el método de la interface IAdaptador:

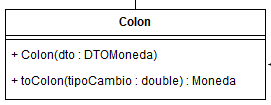
Para realizar la petición se abre una conexión localhost, se obtiene los datos, se cierra la conexión y por último se devuelve el resultado. De esta manera el patrón se cumple y funciona de “traductor”.

Para cumplir Adaptador se siguen los siguientes puntos como estrategia:

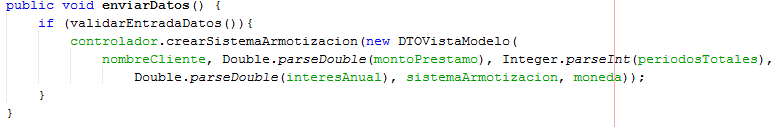
* Al tener clases clientes no compatibles con clases proveedoras, usar una capa intermedia de traductores con el patrón adaptador.
* El controlador no debe de ninguna manera de funcionar como Controlador.
* El Adaptador debe siempre poseer transparencia en ambas direcciones.

## DTO

Esta patrón supone que la comunicación entre las capas debe ser por medio de DTOs (data transfer object) que se pueden definir como “pequeños repositorios de información”. Esta capa es transversal a las demás capas. A nivel de diseño se tiene el siguiente ejemplo del cumplimiento del patrón:



La clase Colon en su constructor tiene definido para la creación de una instancia de esta clase que es necesario recibir un DTOMoneda. Se puede ver cómo se aplica el patrón para constructores de clase.

 Un ejemplo a nivel de código del uso del patrón DTO es el siguiente:

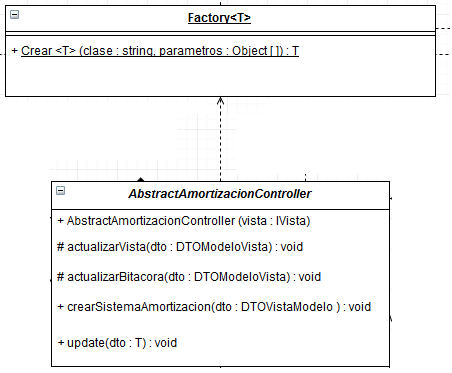
En este caso se utiliza el patrón para ser usado para la comunicación entre la vista y el controlador. La vista no le envía dato por dato al controlador, sino que crea un DTO y se lo manda a este. Se puede notar el cumplimiento de este patrón entre paquetes y cumplir MVC.

Para cumplir el patrón DTO se tuvo la siguiente estrategia:

* Si un método recibe parámetros, meter un DTO (versión extremista).
* La comunicación entre capas siempre debe ser entre DTOs (ya sea como parámetros o valores de retorno).
* Para los constructores de entidades, utilizar DTO.

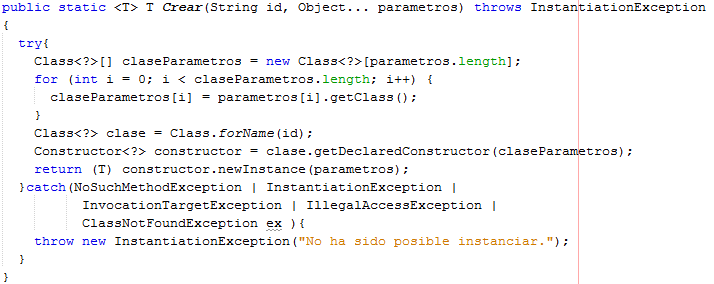
## Factory (creacional)

Se utiliza Factory como patrón creacional. Supone la creación de distintos objetos. Aunque hay muchas variaciones del patrón Factory, se utilizó el siguiente:



En este caso el *AbstractAmortizacionController* es el cliente. La fábrica es enormemente reflexiva ya que solo posee un método: crear que debe recibir el nombre de la clase y los posibles parámetros. Puede crear cualquier entidad del modelo. Es importante realizar validaciones en la interfaz para asegurarse que el nombre de la clase en la entrada de usuario sea correcto y evitar posibles excepciones.

A nivel de código se tiene los métodos de creación:



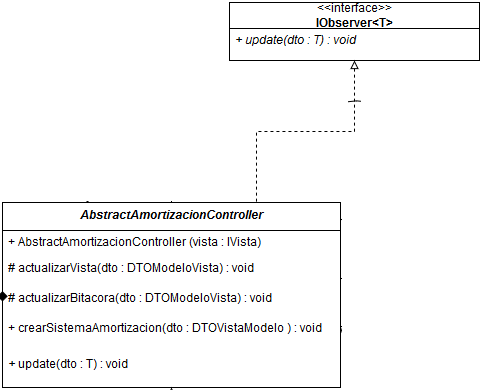
Utiliza reflexión para cumplir el OCP. Devuelve un objeto del tipo correspondiente con sus atributos inicializados. Puede recibir varios objetos que serán usados en el constructor de la clase correspondiente.

Para cumplir con el patrón de diseño Factory se utilizó la siguiente estrategia:

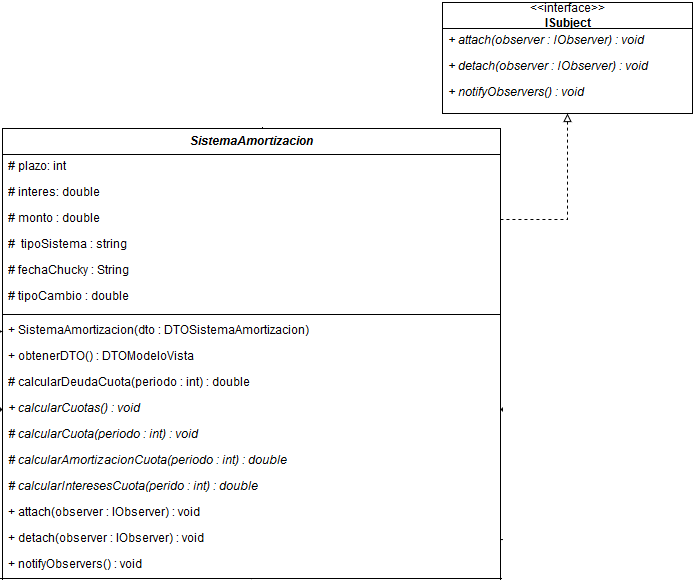
* En los métodos de creación siempre utilizar reflexión.
* Devolver tipos object genéricos para evitar la dependencia entre el Factory y las otras clases.
* Se aprovecha el patrón de diseño cuando se tiene una jerarquía de objetos de los objetos a crear.

## Observador (comportamiento)

Este patrón es usado cuando se tiene un objeto que es observado (subject) y un conjunto de observadores (observers) de manera cuando el subject realiza un cambio en su estado, los observers sean notificados inmediatamente y cambien su estado también. Se tiene el siguiente ejemplo a nivel de diseño:

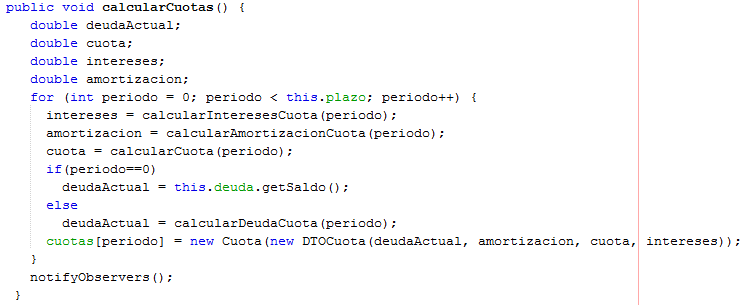


El controlador se vuelve observador del modelo.



Tal implementación es útil ya que modelo siempre hay uno pero pueden haber n controladores. De tal manera que cuando le modelo termine de realizar cálculos, los controladores reciban tal notificación y estos actualicen a las vistas respectivas posteriormente.

A nivel de código se tiene el siguiente ejemplo:



CalcularCuotas() al terminar de realizar todos los cálculos de un sistema de amortización al final tiene el método notifyObservers(), de tal manera que los controladores sean notificados de que ahora pueden comunicarse con la interfaz.

Para cumplir con el patrón de observadores se siguieron los siguientes puntos como heurística:

* Nunca un observador debe poder conocer a los otros observadores.
* Es importante que el subject puede tener cualquier cantidad de observadores.
* Se prefiera usar la versión de observer “push” para reducir el acoplamiento y así los observers no tiene que pedir información al subject cuando estos se actualicen.
* El “linkeo” entre subject y observer nunca se debe realizar en el subject. El subject debe siempre ser totalmente agnóstico a los observers.

# Ejecución de funcionalidades

Se debe incluir:

* Screenshot de la funcionalidad que se está ejecutando al usuario.
* Screenshot del resultado de la funcionalidad.
* Se debe hacer para ambas interfaces (GUI y Consola).
* Además se debe mostrar un Screenshot con la información histórica del contenido del XML y otro para la información histórica del CSV.

Se probarán todas las funcionalidades. Estas corresponden a Consultas de inversiones: corrientes, certificadas y pactadas. A continuación se muestran dichas pruebas.

## Caso 1 : Sistema Amortización Alemán

### Modo GUI

Se tienen los siguientes datos de entrada:

Se obtuvieron los siguientes resultados (Correctos):

Se realiza exactamente la misma prueba y con los mismos datos de entrado pero ahora dolarizados:

### Modo Consola

Se tienen los siguientes datos de entrada:

Se obtuvieron los siguientes resultados (correctos):

Se realiza exactamente la misma prueba y con los mismos datos de entrado pero ahora dolarizados:

## Caso 2 : Sistema de Amortización Francés

### Modo GUI

Se tienen los siguientes datos de entrada:

Se realiza exactamente la misma prueba y con los mismos datos de entrado pero ahora dolarizados:

### Modo Consola

Se tienen los siguientes datos de entrada:

Se realiza exactamente la misma prueba y con los mismos datos de entrado pero ahora dolarizados:

## Caso 3 : Sistema de Amortización Americano

### Modo GUI

Se tienen los siguientes datos de entrada:

Se realiza exactamente la misma prueba y con los mismos datos de entrado pero ahora dolarizados:

### Modo Consola

Se tienen los siguientes datos de entrada:

Se realiza exactamente la misma prueba y con los mismos datos de entrado pero ahora dolarizados:

## Bitácoras

Luego de realizar las pruebas anteriores se tienen las siguientes bitácoras (correctas):

### XML

### CSV