¿Qué es diseñar?

La resolución de un problema conlleva el construir una solución que resultará siendo un sistema de información. Se sabe que el software solo es una rama de un sistema de información.

El análisis del problema me permitirá entender el problema y gracias a ello podré definir los requerimientos adecuados para el diseño de la forma que tendrá la solución.

Es importante definir los **componentes** (objetos, clases, métodos, etc), las **responsabilidades** (que resolverá cada uno de los componentes) y las **relaciones entre ellos** (como interactúan entre sí).

A lo largo del año se optará por elegir ciclos de vida variados, ya que habrá que evaluar el más adecuado para cada situación, pero se realizará un enfoque principal en base al **iterativo e incremental**.

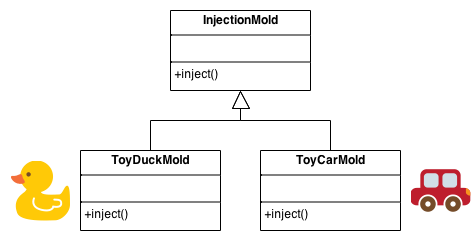
Diseñar entonces consiste en tomar decisiones, evaluar las diferentes alternativas y ponderar las cualidades de cada una de estas.

Patrones de diseño

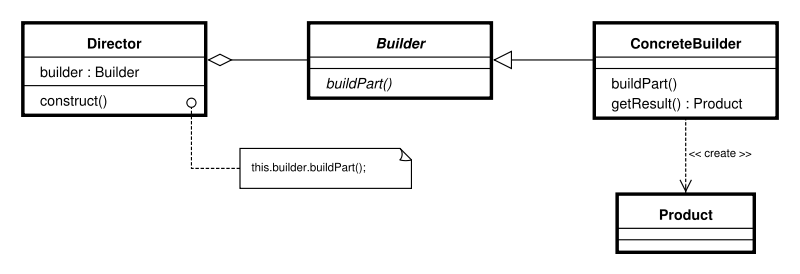
Se denomina patrón de diseño a una solución particular para casos generales. Tienden a ser ideales y tienen cualidades, pero no representan a un fin en sí mismo. Cabe destacar que un patrón de diseño no representa la solución real a ningún problema, sino que es un medio que puede facilitarme la resolución de este (puedo utilizarlo como base y aplicarle modificaciones o no, depende el tipo de problema que este manejando).

**Patrones creacionales**

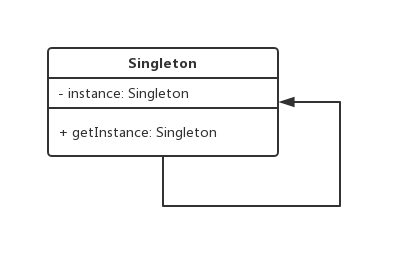
* **Factory** **Method**



* **Builder**: permite construir objetos fácilmente y por partes (por si no conozco todos los atributos necesarios en primera instancia), lo cual garantiza la consistencia de estos.

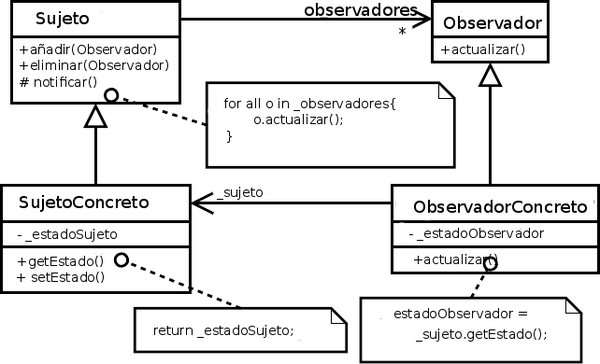


* **Singleton**: consiste en permitir que una determinada clase se instancie por una única vez.



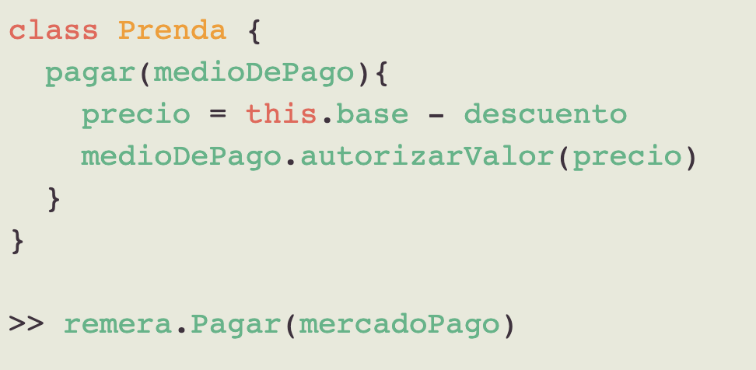
**Patrones de comunicación**

* **Call and return**: un objeto llama a otro y queda esperando hasta que este le devuelve un resultado. No hay efecto de lado.
* **Memoria compartida**: un objeto llama a otro con el cual comparte un espacio de memoria compartida, pero no se queda esperando hasta que este termine sino que sigue su ejecución (luego B puede avisarle que terminó o directamente busca en el espacio de memoria compartida cuando necesite la información que le solicitó a B).
* **Exceptions**: generalmente se utilizan para propagar fallos. La excepción burbujea hasta que alguien de más arriba la atrapa.
* **Callbacks**: un objeto llama a otro y además le envía las “instrucciones” sobre lo que debe hacer con el resultado una vez termine su ejecución. El primer objeto continúa su ejecución.
* **Observer**

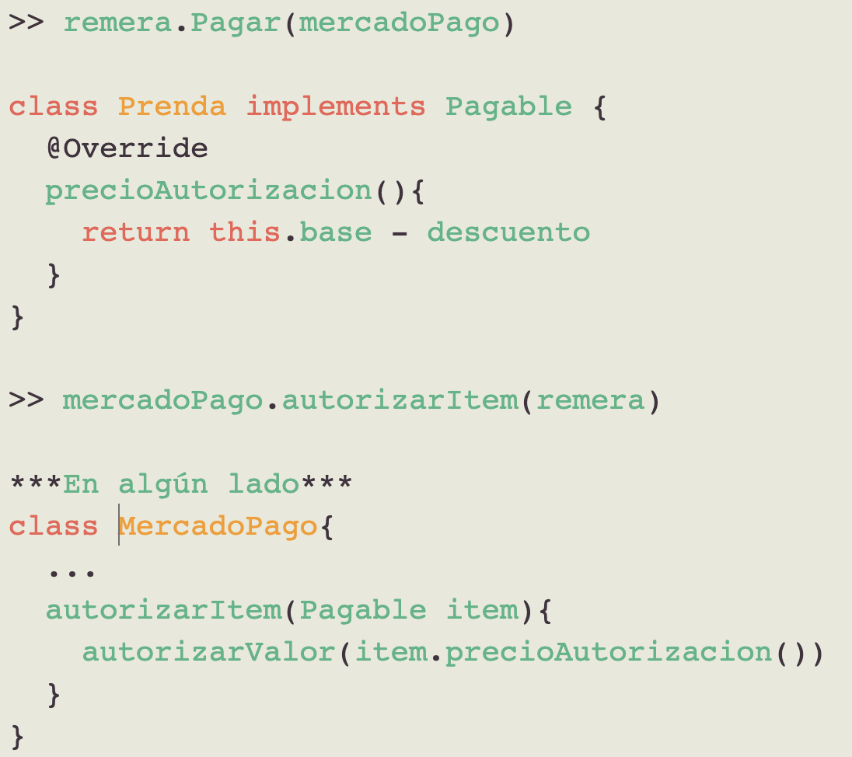


Inversiones de control

* **Control directo**



* **Control inverso**



Sea cual sea la inversión de control que realice, será fundamental saber que habrá un contrato entre los objetos.

Inyección de dependencias

Formas de obtener dependencias (algo que un objeto necesita para su funcionamiento):

* **Singleton**: está muy acoplado a la fuente y es poco testeable.
* **Service Locator**: es más configurable y un poco más testeable.
* **Inyección de dependencias**: el objeto recibe las dependencias mediante el constructor o setters (no conoce la fuente, simplemente lo recibe). Esto hace que el acoplamiento con la fuente sea bajo y facilita el testeo.

Refactoring

Siempre es fundamental aceptar que nuestro sistema tendrá cambios. Ante un cambio podemos reaccionar de varias formas (incluido el no actuar), pero vamos a destacar dos:

* **Parchear**: generalmente es una solución rápida, pero siempre esto será temporal, nunca será una solución estable y sostenible en el tiempo.
* **Refactorizar**: lleva un análisis más profundo para realizar cambios en mi diseño, pero efectivamente los resultados serán sostenibles en el tiempo (si es que fue bien diseñado y no hay cambios rotundos en el futuro).

El refactorizar consiste en adaptar nuestro diseño a los cambios requeridos en un determinado momento cuando noto que mi diseño no los soporta. Es fundamental mantener el ojo en los requerimientos y no perder el enfoque principal. No se relaciona con optimización de código.

Interfaces entre componentes

La interfaz es una forma de conexión entre dos o más elementos. Permite abstraerse de detalles internos de cada uno de los elementos y genera abstracciones de utilidad.

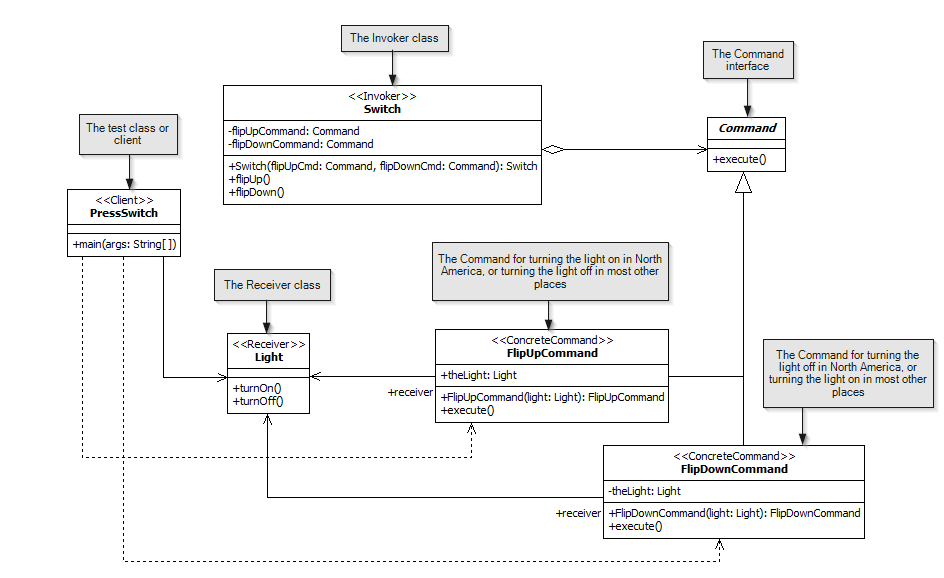
* **Interfaz saliente**: como quiero hablarle al mundo exterior.
* **Interfaz entrante**: como quiero que el mundo me hable a mí o como el mundo quiere que yo le hable.

Surge el concepto de adaptador para poder adaptar una API externa a mi interfaz entrante por ejemplo.



Reificación de comportamiento

Consiste en la cosificación de diversos comportamientos. Es posible entenderlo de forma más sencilla a través del patrón Command.



Tareas periódicas

Se tendrá en cuenta la **planificación externa** únicamente. Se necesitarán para esta: un punto central de acceso, un método main, el assembly plugin para armar los package y la configuración del crontab.