Resumen epiquisimo para

Final de

Orientación a Objetos 2

Temas

* Closures
* Seaside
* Refactoring
* Testing
* Patrones
* Framework

**Closures o clausuras léxicas**

Son aquellos pedazos de código que pueden ser usados en distintos contextos y pueden ser enviados como parámetro. Permiten generar nueva funcionalidad sin que realmente se esté cambiando la estructura del objeto que utiliza dicho bloque (closure), por ejemplo, en button se puede enviar un closure que se utilice en el onClick (actionBlock en smalltalk). O para callbacks, es decir, cosas que deben ejecutarse una vez terminado el procesamiento de una función asincrónica.

* Una expresión entre corchetes crea un bloque.
* Los bloques pueden ser enviados como parámetro, pueden ser asignados a variables y pueden recibir parámetros.
* Un bloque puede hacer **referencia** (no valor), es decir, usar variables del contexto donde estén. Por ejemplo, si se las llama en un método, pueden usar variables temporales del mismo, o inclusive variables internas de la instancia (variables de instancia).
* Una vez creados no se pueden modificar, esto pasa porque se tienen referencias al bloque, por lo tanto si se modifica, en realidad se estaría creando uno nuevo. Es como pasa con los strings, los bloques serían inmutables.
* **Complican la depuración** y hacen los programas más difíciles de entender, así que si se puede lograr lo mismo de otra manera es mejor no usarlos.

**Seaside**

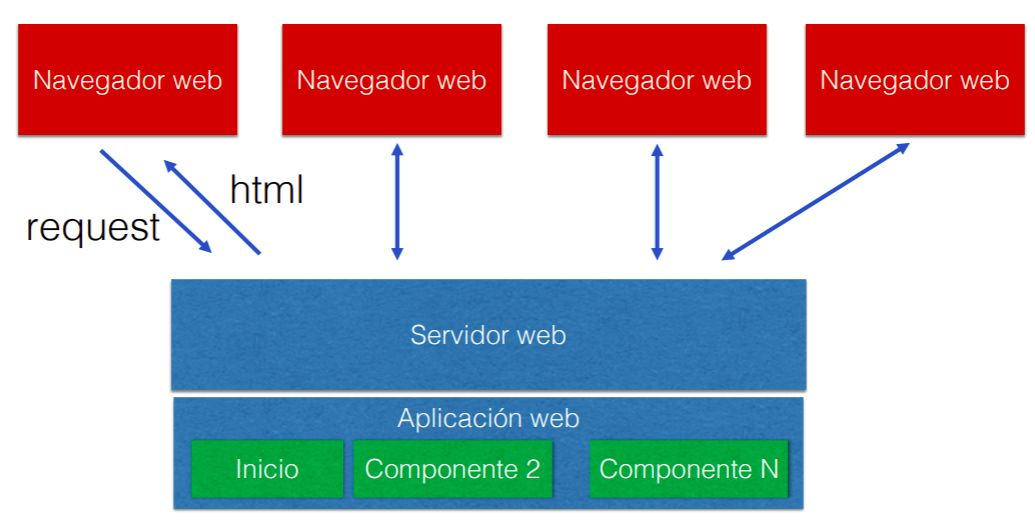
Es un framework para construir aplicaciones web en smalltalk.

Consiste en un conjunto de componentes conectados y reutilizables (como React).

Incluye su propio servidor de aplicaciones, que además mantiene el estado de la sesión.

Puede generar y devolver HTML válido.

**Anatomía simplificada de una app web de seaside**

****

Cuando un cliente accede al servidor, se genera un pedido que es atendido por este último el cual deriva el pedido a un objeto que lo atienda y construya la vista (html) que se requiera. Por cada primer pedido, el servidor crea una sesión para el cliente.

Pasos para hacer una app en seaside

1) Creo una subclase de WAComponent para el componente principal. (inicio) de mi aplicación.

2) Implemento el método #renderContentOn: en ese componente.

3) Registro el componente para que atienda los pedidos para mi aplicación (WAAdmin, método register y asAplicationAt(url).

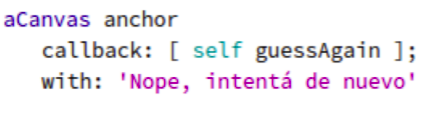
El WAAdmin register no genera instancias del componente que se muestra a primera vista.

Cuando un usuario entra al link, recién ahí se crea una instancia nueva del componente que se describió WAAdmin al registrarlo.

Entonces, los componentes se crean cada vez que un usuario abre la página, y por cada sesión.

Pinceles:

* paragraph with: “Mensaje” o bloque →párrafo
* strong with: “Mensaje” o bloque →negrita
* link:



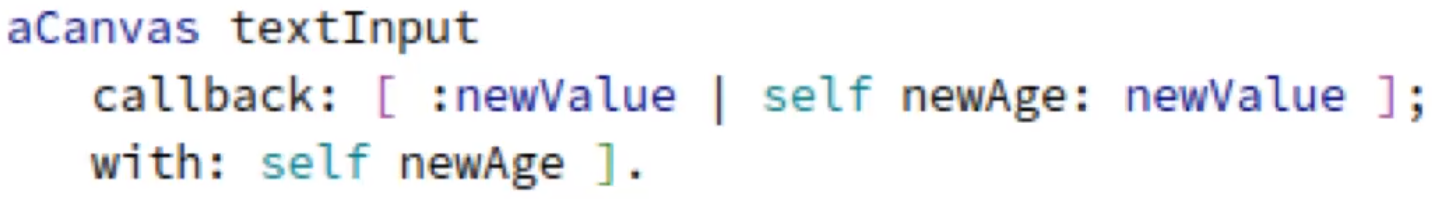
En el callback se utiliza un closure que llama a un método propio del componente subclase de Component.

En seaside, se tienen los métodos call y answer heredados de WAComponent los cuales indican el camino a seguir entre los subcomponentes del mismo.

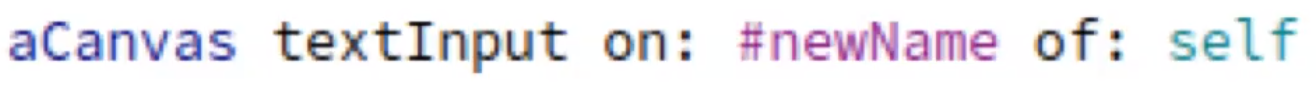
call lo que hace es ceder el manejo de la aplicación al componente que se paso por parámetro (self call: miComponente) y frenar su ejecución en ese momento. El otro componente despliega su vista y hace lo que debe hacer y luego con el mensaje answer (self answer) devuelve la ejecución al punto siguiente al call.

Hay dos formas de configurar un input:

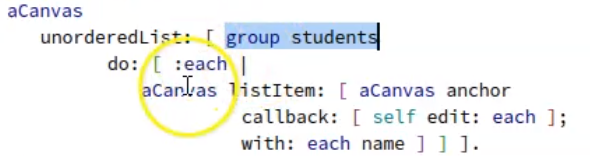
* Con with y callback: Indicando qué hacer cuando se cambia el valor y cual mostrar.



* Con selector, indicando el nombre del campo y este debe tener los get y setter correspondientes (como si fuera un javaBean).



Existen algunos pinceles que no necesitan poner el with: sino que directamente se hace dos puntos y se le envía el bloque que representa su contenido, ej:



No se tiene un with: en unorderedList ni en listItem.

El renderContentOn se llama cada vez que se recarga la página.

No se puede enviar mensajes al canvas dentro de un callback justamente porque primero se ejecuta el callback y luego se recarga la página, por ende, el canvas ya no es el que tiene asociado el callback (es uno nuevo, el callback tiene un canvas viejo).

**Inversión de control**: es cuando algún componente del framework le manda mensajes a los componentes que crea el programador.

**Framework de caja blanca**: son aquellos framework en los cuales se tiene disponible el código fuente y se puede mirar para saber usarlos bien.

**Código de dominio vs Código de presentación**: básicamente es tener código de backend y de frontend separados. Las clases que incluyan algún manejo de interfaz son de código de presentación y no deben hacer cosas referidas a la manipulación de datos, a excepción de alguna que otra transformación del dato para poder mostrarlo, pero nada más.

Por otro lado, la clase que pertenezca al código de dominio no puede incluir ningún tipo de código que toque la interfaz.

**Sesiones**

Lo que se pretende es que cada vez que se crea un componente nuevo para atender a un cliente, dicho componente consuma de la misma colección que se generó al principio y que de esa forma, a pesar de estar en distintas sesiones, se pueda tener lo mismo que tienen los demás.

Como en smalltalk, las clases también son objetos, se pueden definir variables de instancia en éstas.

En el caso de que se tenga una clase de objeto particular y siempre que se genere una instancia se quiera tener la misma instancia para cualquiera que lo genere, entonces se puede recurrir a un método de clase.

En dicha clase se va a generar una variable de clase que tenga en ella la instancia de la clase que quiera mantenerse igual para todos.

En un método aparte del new (porque luego se necesita) como podríamos llamarle “soleInstance”, se va a inicializar la variable anteriormente mencionada con una instancia del objeto si es que este objeto es nil (por esto no usábamos el new, porque se necesita para inicializarlo), luego de esto se devuelve la instancia.

Si no era nil simplemente se devuelve.

También es bueno tener un método que resetee la variable en nil por si así se lo desea.

Estos métodos de inicializar solo si es nil y para limpiar la variable, se llaman **inicialización haragana**.

**Yourself**

Al enviar yourself a lo último de una cascada, se asegura de que lo que se devuelve al terminar la cascada es el primer objeto de la misma.

Ej:



En este caso, el objeto que se va a enviar como parámetro en call va a ser ForumComponent, dado que es el primer objeto que aparece en la cascada. Con esto evitamos que se mande aForum sin querer.

**Pasando modelos**

Los componentes de interfaz necesitan un modelo de dominio para mostrar. El que se encarga de ofrecerles este dominio es justamente el componente que hizo el call, es decir, el que lo llamó. Al llamarlo se le enviará el objeto que necesita el componente, y dicho objeto está relacionado al link que se clickeo para llamar al componente. Por ejemplo, un curso que tiene links hacia distintos foros, en cada link tiene asociado una instancia del objeto de dominio aForum distinta.

Este componente de curso le pasa el foro al componente de foro, y con esto se resuelve dicho problema.

Pero hay otra complicación, cuando se quiere tener un objeto en todos los componentes, ej: el usuario.

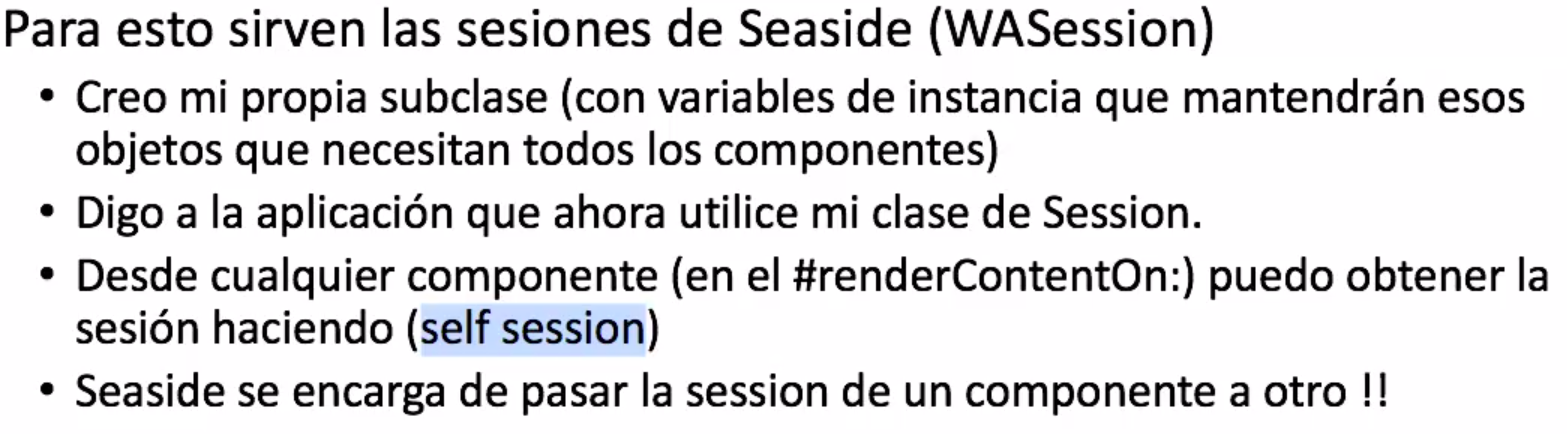
Para no andar pasando al usuario por parámetro cada vez que se llama a un objeto, se puede utilizar un objeto que brinda smalltalk, WASesion.

Se puede generar una subclase de WASesion que tenga todas las variables de instancia que se necesita que sean común al resto de componentes, como el usuario.

Entonces al registrar la aplicación también se le indica que utilice la subclase de WASession que se generó anteriormente.

Finalmente, en cada componente, cada vez que se requiera al usuario, simplemente se hace self session y con esto se obtiene la instancia de WASesion que se creó al principio, y a este se le puede pedir el usuario.

**Info de la diapositiva:**

****

**Testing**

Lehman tiene varias leyes que tienen que ver con los cambios que se van produciendo en un sistema y cómo afecta al mismo, se rescatan 4 para la materia.

* Continuing Change(Cambio continuo): Los sistemas deben adaptarse continuamente o se vuelven progresivamente menos satisfactorios
* Continuing Growth (Crecimiento continuo): La funcionalidad de un sistema debe ser incrementada continuamente para mantener la satisfacción del cliente.
* Increasing Complexity (Complejidad creciente): A medida que un sistema evoluciona su complejidad se incrementa a menos que se trabaje para evitarlo.
* Declining Quality (Calidad en declive): La calidad de un sistema va a ir declinando a menos que se haga un mantenimiento riguroso

**Pérdida de negocio**

Dado que el ritmo de los cambios producidos en los negocios va incrementando exponencialmente justamente el tiempo para reaccionar a dichos cambios también disminuye exponencialmente, entonces se pierden negocios.

Si no se puede mantener un ritmo constante de adaptación, entonces el negocio se quiebra.

**Mantenimiento**

Existen varios tipos de mantenimiento: correctivo, evolutivo, adaptativo, perfectivo y preventivo.

Sin embargo, a día de hoy se pasa el 50% de tiempo intentando leer y entender el código anterior.

Entonces la incapacidad de poder hacer código más rápido hace que se pierdan oportunidades de negocios.

Esto sucede por lo llamados **Big Ball of Mud**.

**Big Ball of Mud (BBoM)**

Son aquellos códigos que no son legibles ni flexibles en absoluto, hasta a veces llegan a dar dolor de cabeza entenderlos.

Si bien funcionan no son fáciles de comprender.

A veces se empieza generando este tipo de código ya que no se entiende muy bien el dominio de la aplicación, y una vez formulada la misma se comienza a percibir que habría otras maneras mucho mejores de solucionar los problemas, es aquí donde se necesita lo que se llama *refactoring*.

**Patrones de desarrollo de código**

Existen varios patrones que se dan a la hora de desarrollar código que podría provocar BBoM.

* *Throwaway Code*: Es aquel código que fue diseñado para desecharse, como prototipo, pero en algún punto termina quedando como código oficial simplemente porque funciona.
* *Piecemeal Growth*: Se comienza con un código bien estructurado y con buena arquitectura pero surgen cambios, bugs, fixes, requerimientos, cambios tecnológicos, etc; y se tiene que ampliar y/o modificar el código existente logrando que se corroa la buena estructura que había antes.

**Refactoring**

Los elementos distintivos de la arquitectura de un sistema no surgen hasta después de tener código que funciona, es importante tener en cuenta el feedback que proporciona el código existente y el funcionamiento actual de la aplicación para poder realizar un refactoring de la misma.

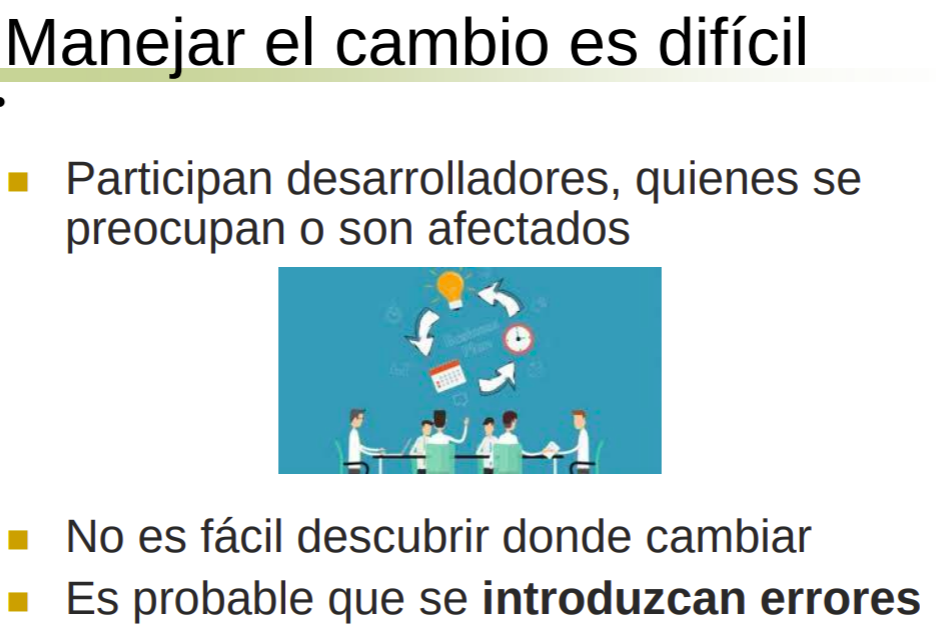
El refactoring es una transformación que preserva el comportamiento, pero mejora el diseño, organización, legibilidad, adaptabilidad y mantenibilidad del código.

Implica:

* Eliminar código duplicado.
* Simplificar lógicas complejas
* Clarificar código (hacer que sean más legibles)

Cuando:

* Una forma seria realizar refactoring una vez construido el código y que pasen los test (de esta manera se puede tener un feedback y una manera de comparar el código).
* Otra manera y por la mejor, es ir haciendo refactoring a medida que se va desarrollando el código, cuando se encuentra código feo y difícil de entender o cuando se necesita una reorganización para implementar otro cambio.
* O antes de llegar a los famosos BBoM.



La importancia del refactoring reside en que:

* Es nuestra única defensa contra el deterioro del software.
* Facilita la incorporación de código nuevo.
* Permite preocuparse por la generalidad mañana. Es decir, permite ser ágil en el desarrollo.

**Test Driven Development**

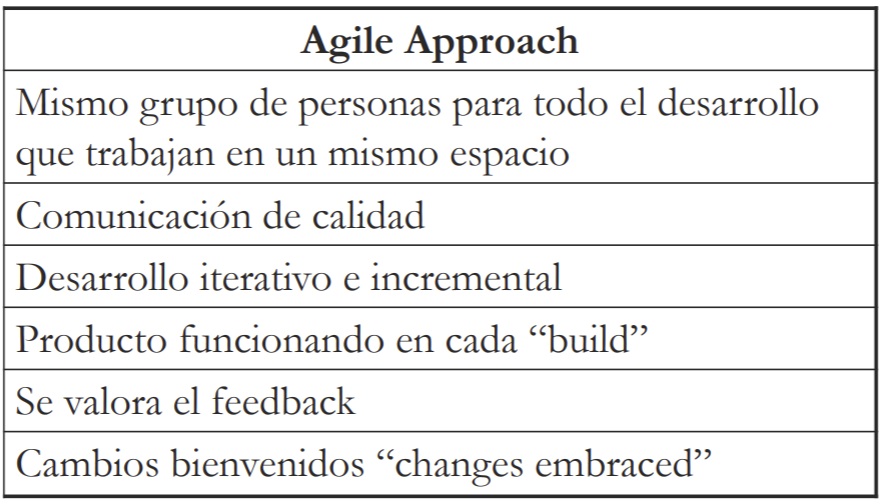
Antes de la aparición de TDD se tenía al testing como un concepto que se realizaba luego de generar el código, es decir, a lo último.

Se tenía un equipo aparte, QA (quality assurance), que se encargaba de generar estos test y la documentación correspondiente para ejecutarlos, y además luego de hacerlos se le llevaba una lista de bugs, resultado del análisis, a los programadores.

Esto provocaba que se tarde más, dado que eran bugs con los cuales los programadores o bien no estaban familiarizados con el código o ya eran muy viejos y estaban ocupados en otra cosa.

En la ingeniería de software se comenzó a desarrollar lo que se conoce como metodologías ágiles que proponen un par de manifiestos.

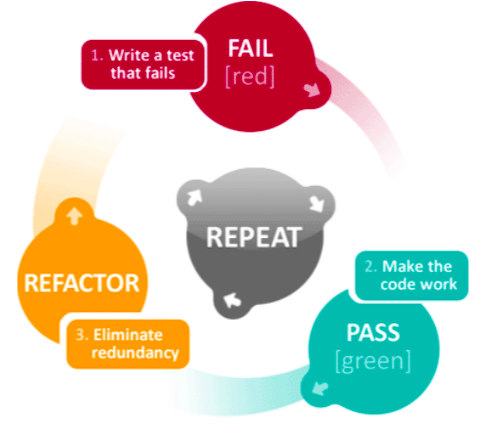


Esta metodología apuntaba a ser adaptativa y orientada a las personas, por esto tenía un par de características:

A partir de este concepto de ir trabajando de partes, no se tenía una idea de cómo guiarse para desarrollar sin tener un modelo o diseño completo.

Por esto, Kent Beck, propone el método Test Driven Development, el cual reside en generar los test antes del código.

Como era de esperarse, dichos test fallaban, pero justamente se partía desde allí para generar código que haga que funcionen. Una vez logrado esto, se seguía a la parte de refactoring para luego utilizar estos test y asegurarse de que el comportamiento no cambia con la refactorización.



TDD combina Test First Development y Refactoring.

**Tipos de test, granularidad**

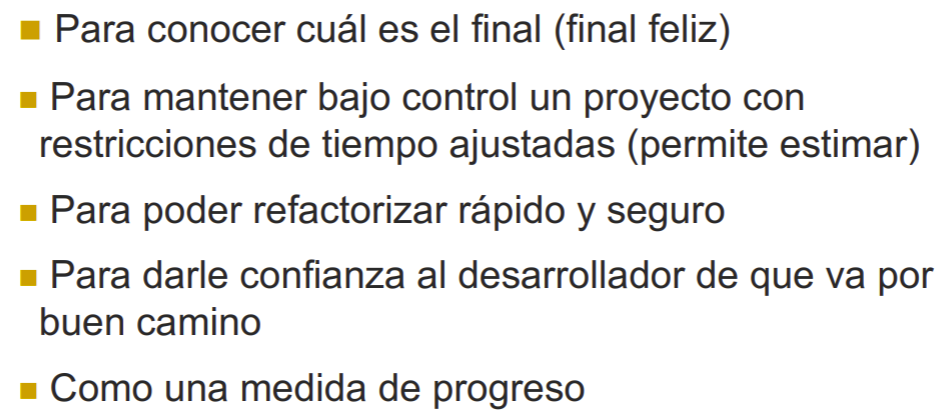
**Test de aceptación**:

* Por cada funcionalidad esperada.
* Escritos desde la perspectiva del cliente.

**Test de unidad**:

* Aislar cada unidad de un programa y mostrar que funciona correctamente.
* Escritos desde la perspectiva del programador

**Razones para no dejar el testing para el final**



**Algunas reglas de TDD**

* *Diseñar incrementalmente*: teniendo código que funciona como feedback para ayudar en las decisiones entre iteraciones.
* *Los programadores escriben sus propios tests*: no es efectivo tener que esperar a otro que los escriba por ellos.
* *El diseño debe consistir de componentes altamente* ***cohesivos*** *y* ***desacoplados*** *entre sí*: mejora evolución y mantenimiento del sistema.

**Automatización para TDD**

Se necesita tener herramientas que automaticen el testing (ej: xUnix) y además no deberían tardar más de 10 minutos en proveer una respuesta. Sin estas cosas, implementar TDD es difícil o imposible.

**Que se logra con TDD**

* Diseño simple y limpio
* Desarrollar más rápido
* Saber cuando terminamos
* Confianza para el desarrollador
* Coraje para refactorizar
* Documentación práctica que evoluciona naturalmente
* Incrementar la calidad del software: que el código se construya correctamente y que el código construido sea correcto.

**Problemas y soluciones con TDD**

Problemas:

Unit testing infinito: por cada método público (muy costoso).

Test coupling: al estar los tests atados a la implementación. (es decir, que solamente funcionan si un método tiene un nombre en particular o pertenece a esa clase en particular).

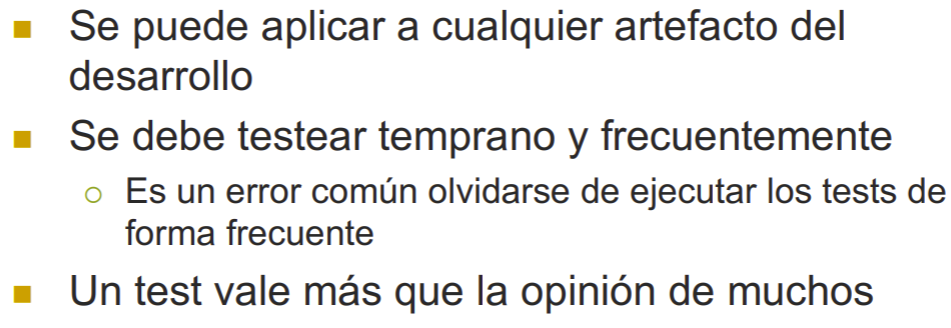
Soluciones:

“Test with a purpose” (Kent Beck), es decir, saber por qué se testea algo y a qué nivel debe testearse. El objetivo de testear es encontrar bugs, entonces se debe testear tanto como sea el riesgo del artefacto.

El riesgo puede reducirse de distintas formas:

* reduciendo la cantidad de bugs.
* previniendo la aparición de bugs.
* acercando el SUT(Sistema a testear, System under testing) a las necesidades del usuario
* testeando el SUT bajo condiciones extremas

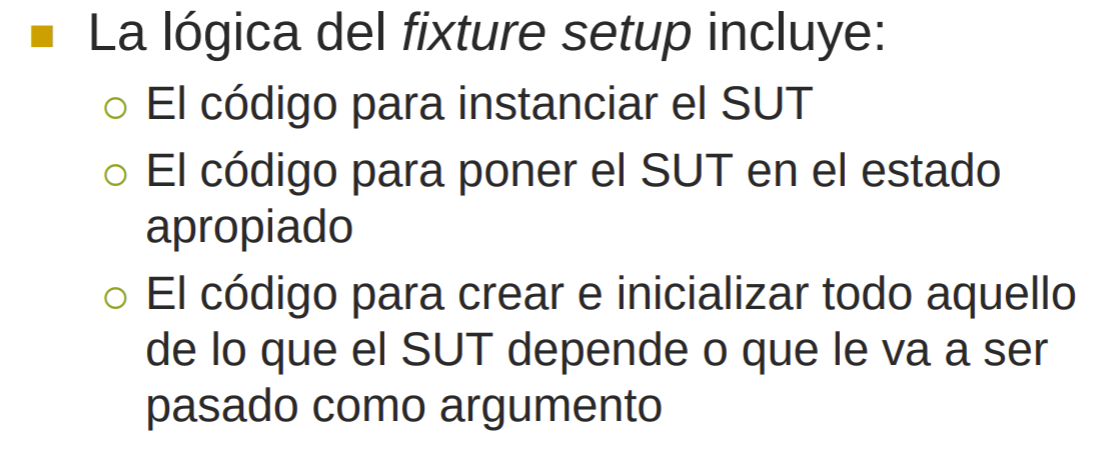
**Como, cuando, porque testear**



**Tamaños de testing**

Una forma es un testing por cada condición, es la forma más purista, se puede detectar los errores más fácilmente dado que no se tienen múltiples caminos.

Sin embargo es más complejo dado que el costo de implementarlo es más grande y no es posible tener una cobertura al 100%. No se sabe a ciencia cierta si realmente vale la pena.



**Aislar el SUT**

Cuando se testea un objeto o clase en particular, a veces se necesita la ayuda de otras clases de las cuales depende el sut. Sin embargo, no es el objetivo testear estas clases sino el sut particular, pero como se lo necesita a la hora de testear, estos deben ser instanciados y pueden acarrear problemas propios de dichos objetos.

Es aquí donde entran en juego los Test Doubles.

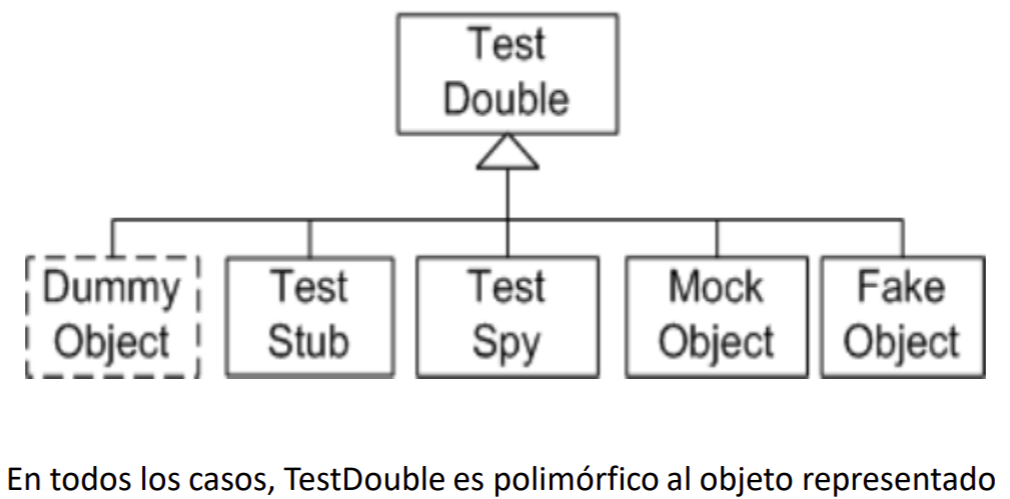
**Test Doubles**

Estos son objetos que simulan ser los objetos que necesita el sut para evaluarse, es decir, simulan su comportamiento.

Puede ser que los módulos de los cuales depende el sut no puedan utilizarse en el ambiente de prueba por ciertas razones:

* No está implementado
* No se puede acceder
* No se puede o es muy difícil replicar
* Es un objeto complejo que puede tener errores en sí mismo que no quiero acarrear

**Tipos de test doubles**



**Dummy object**: se utiliza el objeto para que ocupe un lugar pero nunca es utilizado

**Test Stub**: sirve para que el SUT le envíe los mensajes esperados, y devuelva un valor por defecto

**Test Spy**: Test Stub + registro de outputs (recibe la salida del sut)

**Mock Object**: test Stub + verification of outputs (recibe la salida del sut y lo verifica o “imita” lo que haría el verdadero objeto)

**Fake Object**: imitación. Se comporta como el módulo real (protocolos, tiempos de respuesta, etc)



**Cuando usar test doubles**

El momento indicado para usar test doubles es cuando el objeto real es un objeto complejo que:

* Retorna resultados no-determinísticos,es decir, no es posible determinar en el momento del testeo (ej., la hora actual o la temperatura actual).
* Tiene estados que son difíciles de reproducir (ej., un error de network).
* Es lento (ej, necesita inicializar una transacción a la base de datos).
* Todavía no existe (puede pasar al usar la técnica TDD).
* Tiene dependencias con otros objetos y necesita ser aislado para testearlo como unidad.

**Reglas del testing**

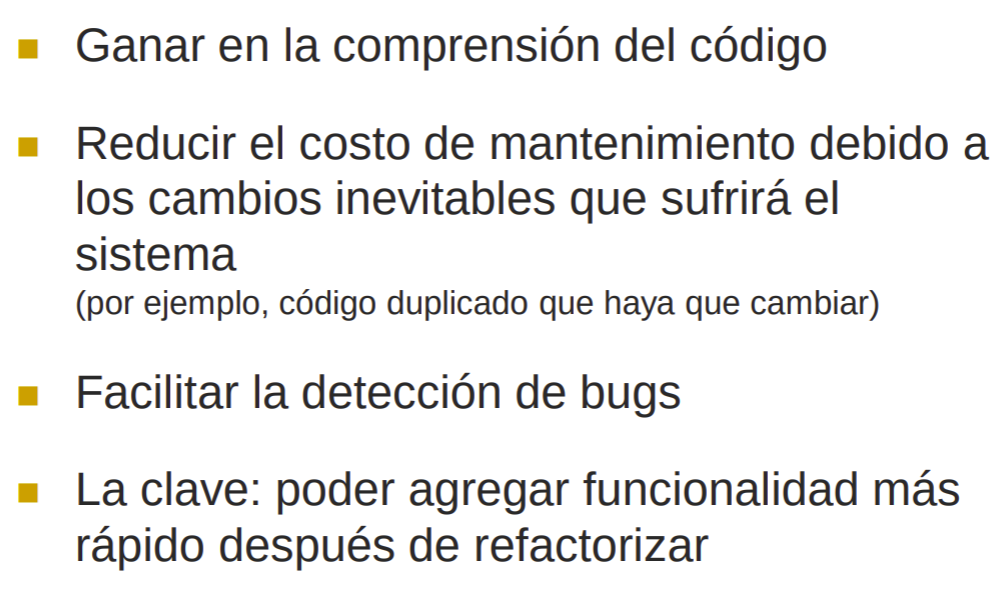
* Mantener los tests independientes entre sí (un test no puede depender de otro test).
* Un buen test es simple, fácil de escribir y mantener (que requiere mínimo mantenimiento a medida que el sistema evoluciona, si es costoso no vale la pena, a menos que sea menos costoso que el error provocado).
* El objetivo de testear es encontrar bugs.
* Limitaciones del testing: no encuentra todos los errores no puede comprobar la ausencia de errores.

**Refactoring**

Lo que se quiere lograr es que ante un cambio se pueda saltar a un único punto en el sistema a realizar al cambio.

El tema de performance es mejor dejarlo para después del refactoring, es decir, una vez refactorizado optimizar el código.

**Importancia del refactoring**



**CLEAN Code**

CLEAN:

* **C**ohesive(que los métodos y variables tengan sentido y estén conectados entre ellos dentro de una misma clase)
* **L**oosely coupled(menos acoplado, es decir, si se cambia uno no es necesaria cambiar varios)
* **E**ncapsulated (solo se puede acceder a través de métodos)
* **A**ssertive (que tenga asserts, es decir, tests)
* **N**on-redundant(que no haya código duplicado)

La profe agrega **LEGIBLE**

**Bad smells**

Son aquellos indicios de problemas que requieren la aplicación de refactorings.

***Bloaters***

1. **Large Class or God Class**

Es aquella clase que tiene muchos comportamientos asociados, muchos métodos y variables que podrían estar en otras clases, por lo tanto hay baja cohesión en ella.

Problemas: son muy grandes para mantener, pueden generar envidia de atributos, mala asignación, código duplicado, etc.

Solucion: extract method, move method, *Extract Class, Extract Subclass*

1. **Long Method**

Un método tiene muchas líneas de código, el doble de lo normal en un método según el lenguaje que se use.

Problemas: se vuelve ilegible por lo tanto difícil de mantener, cambiarlo y reusarlo. Suele venir acompañado de muchos malos olores como envidia de atributos o código duplicado o simplemente tiene partes de código que podrían ir a un método nuevo.

Solucion: extract method, move method, *Decompose Conditional, Replace Temp with Query*

1. **Long Parameter List**

Es un método con una larga lista de parámetros.

Problemas: Es más difícil de entender. Además también es difícil obtener todos los parámetros para pasarlos en la llamada entonces el método es más difícil de reusar. La única excepción es cuando no se quiere crear una dependencia entre el objeto llamador y el llamado.

Solucion: *Replace Parameter with Method, Preserve Whole Object, Introduce Parameter Object*

**Couplers**

1. **Feature Envy**

Es aquella clase que utiliza los datos y métodos de otra clase para realizar su trabajo.

Problemas: Es un problema de diseño, los datos y las acciones sobre estos deberían estar en la clase que los almacena.

Solución: move method

1. **Middle man**

Se dice que una clase es un Middleman (intermediario), cuando la mayor parte de sus métodos se limitan a delegar en otra clase.

Problemas: Como puede pasar con la generalidad especulativa, están al pedo.

Solución: Remove Middle man

1. **Inappropriate Intimacy**

Significa que existe una dependencia grande entre dos clases, puede darse envidia de atributos.

Problemas: Es difícil hacer un cambio en una clase que no afecte a la otra.

Solución: Move Method/Field

1. **Cadena de mensajes**

Esto sucede cuando un objeto navega a través de una cadena de objetos para obtener un determinado valor.

Problemas: cualquier cambio en la configuración de las relaciones entre los objetos puede romper el código.

ej: (banco cuentaNro: unNro) movimientos first fecha

Solución: *Hide Delegate* (en alguno o varios puntos, esconder mediante un método las relaciones intermedias para llegar al resultado, *Extract Method & Move Method* (es decir, extraer y mover el código que realiza la búsqueda de un objeto a otra clase adecuada y así sucesivamente hasta que sea necesario).

**Dispensables**

1. **Duplicate Code**

El mismo código, o código muy similar, aparece en muchos lugares.

Problemas: hace el código más largo, arreglar un bug en uno de los duplicados no es fácil de extender a los demás duplicados y esto provoca que sea difícil de mantener.

Solución: extract method, *Pull Up Method, Form Template Method*

1. **Data Class**

Es aquella clase que tiene variables de instancia y métodos únicamente para acceder a dichas variables.

Problemas: por lo general estas clases generan que otras clases que interactúan con ella tengan envidia de atributos, es decir, la clase de datos no tiene algunos métodos de los que debería encargarse.

Solución: move method

1. **Generalidad especulativa**

Es cuando se crean clases o métodos donde los únicos clientes son los test. Esto sucede porque se cree que dichas clases se van a utilizar en el futuro y se intenta adelantar a ello.

Problemas: Básicamente carece de sentido su existencia, en criollo, están al pedo.

Solución: Collapse Hierarchy (colapsar las subclases a la superclase y desaparecer la jerarquía) , Inline Class, Remove Parameter

**Tool abusers**

1. **Switch Statements**

Es cuando las sentencias condicionales contienen lógica para diferentes tipos de objetos.

Problemas: la misma estructura condicional aparece en muchos lugares.

Solucion: Replace Conditional with Polymorphism

1. **Legado rechazado (Refused bequest)**

Es aquella clase que hereda de una superclase pero rechaza su legado, es decir, no usa ni datos ni los métodos heredados o estos últimos los redefine.

Problemas: es probable que la jerarquía esté mal diseñada.

Solución: Push Down Method/Field (creo que se refiere a bajar el metodo que se haya en la jerarquia a la subclase que lo necesita)

*Replace Inheritance with Delegation* (cambiar la relación de jerarquía por relegamiento.

**Change preventers**

1. **Cambios divergentes (Divergent Change)**

Esto significa que se implementan cambios distintos en una misma clase, es decir, cambios que no son cohesivos entre sí.

Problemas: Una misma clase tiene comportamientos que no son cohesivos entre sí.

Solución: Extract Class (dividir una clase en diferentes clases).

1. **“Shotgun surgery”**

Cada vez que se realiza un cambio en una clase, se tienen que hacer muchos pequeños cambios en un montón de clases distintas.

Problemas: Es difícil establecer un cambio en una clase sin que afecte a otras, está muy acoplado.

Solución: *Move Method/Field (*es decir, acercar aquellos atributos o métodos que están relacionados muy cercanamente y ponerlos juntos).

*Inline class* (propuesto por la diapositiva en inglés, significa que se junten dos clases en una sola clase)

**Otro “mal olor”**

1. **Comentarios**

Este no es un mal olor en sí, pero depende de lo que se esté comentando puede significar uno. Si se está explicando cómo funciona el código porque no es legible y no se entiende viéndolo entonces podría haber varios malos olores en el código. Uno de ellos podría ser código duplicado o método largo, etc.

Problemas: lo mencionado arriba

Solución: Extract Method, Rename Method (dar un buen nombre a los métodos o variables para que se entienda mejor).

**Refactorings - Catalogo Fowler**

Nota para todos los refactoring, luego de aplicarlos, compilar y testear.

**Composición de métodos**

Permiten “distribuir” el código adecuadamente. Los métodos largos son problemáticos.

1. **Extract Method**

Motivación: métodos largos, métodos muy comentados, incrementar reuso, incrementar legibilidad.

Mecanica: crear un nombre de método que sea autoexplicativo, copiar el código a extraer, revisar las variables:

* Es local y solo se usa en el código extraído: se mueve la declaración.
* Es una variable modificada en el código extraído: si es solo una se trata como query (*replace temp with query*), si hay más de una entonces no se puede extraer el código.

Pasar como parámetro las variables que el nuevo método lee.

1. **Inline Method**

Juntar dos métodos en uno solo, como el inline class.

1. **Replace Temp with Query**

Motivación: Para evitar métodos largos (las temporales, al ser locales, fomentan métodos largos), para poder usar una expresión desde otros métodos o antes de un Extract Method, para evitar parámetros innecesarios.

Mecánica: Primero se busca aquellas variables temporales con una sola asignación (por ejemplo una variable de conteo que solamente se suma en un lugar del código).

Extraer el lado derecho de la asignación hacia un nuevo método y reemplazar todas las referencias de la temporal por un llamado a dicho método. Eliminar la variable.

1. **Split Temporary Variable**
2. **Replace Method with Method Object**
3. **Substitute Algorithm**

**Mover aspectos entre objetos**

Ayudan a mejorar la asignación de responsabilidades.

1. **Move Method**

Motivación: Un método está usando o usará muchos servicios que están definidos en una clase diferente a la suya (Feature envy).

Mecanismo: Mover el método a la clase donde están los servicios que usa. Convertir el método original en un simple delegación o eliminarlo.

1. **Move Field**

Algo parecido al move method pero con field.

1. **Extract class**

Es como el extract method, sólo que en este caso se extrae una clase y se llevan métodos y fields, suele pasar en clases dioses.

1. **Inline Class**

Es convertir dos clases en una sola.

1. **Remove Middle Man**

Es remover la clase que está actuando como intermediario y linkear las clases extremas como debería haber sido.

1. **Hide Delegate**

Es ocultar las clases y métodos involucrados en la obtención de un dato creando un método.

**Organización de datos**

Facilitan la organización de atributos.

1. **Self Encapsulate Field**
2. **Encapsulate Field / Collection**
3. **Replace Data Value with Object**
4. **Replace Array with Object**
5. **Replace Magic Number with Symbolic Constant**

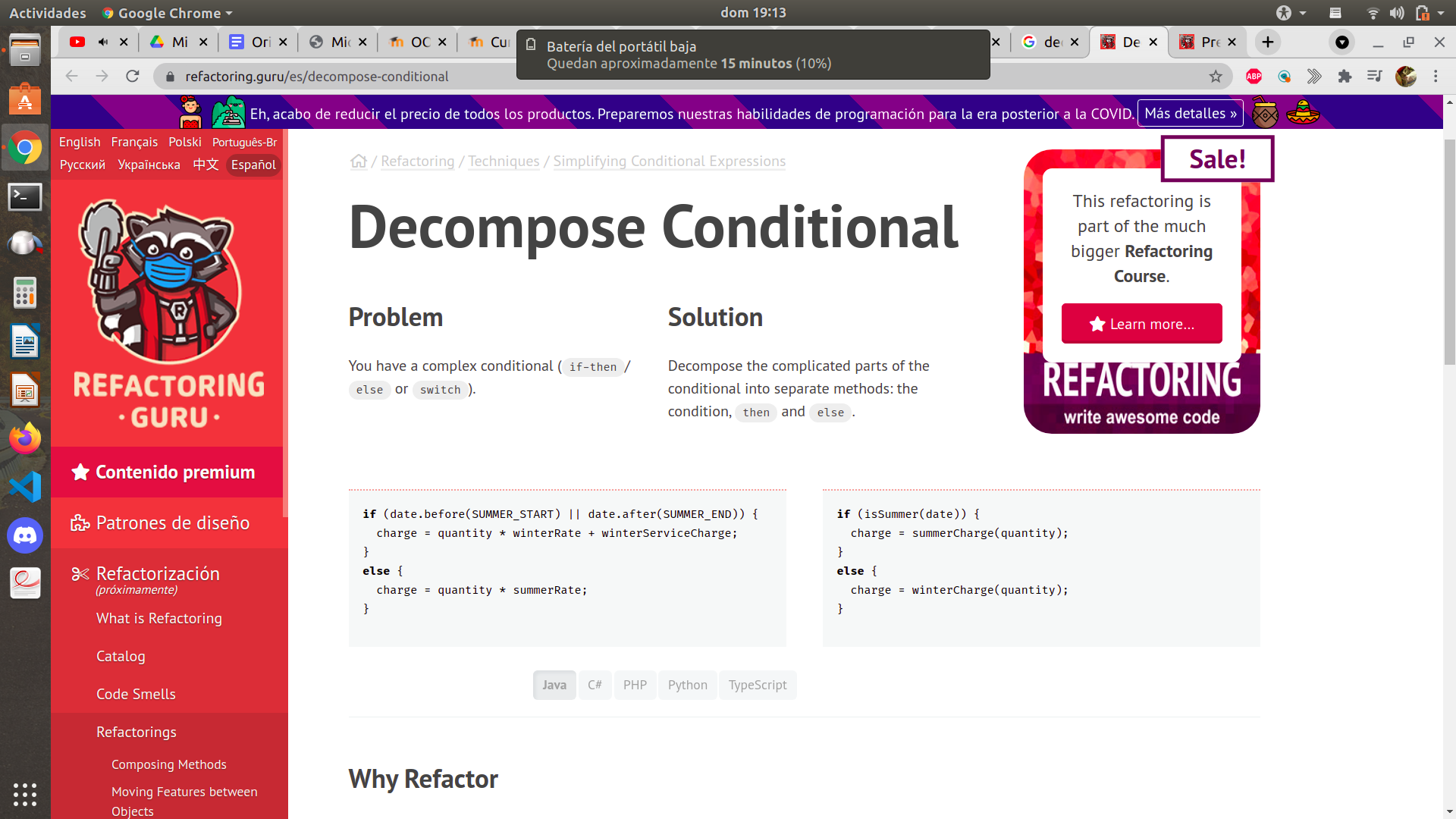
**Simplificación de expresiones condicionales**

Ayudan a simplificar los condicionales.

1. **Decompose Conditional**

Motivación: generar un código más legible en un condicional.

Mecanismo: si se tiene un if y else muy complejos, es decir, se evalúan muchas cosas en el if, y dentro de las instrucciones a aplicar se hacen muchas cosas, entonces mejor extraer estos códigos en métodos y reemplazar por llamadas.



1. **Consolidate Conditional Expression**
2. **Consolidate Duplicate Conditional Fragments**
3. **Replace Conditional with Polimorfism**

Motivación: evitar los condicionales anidados o switches para no tener duplicación de código y la misma estructura en varios lados. Es difícil realizar un cambio porque hay que propagarlo.

Mecanismo: crear la jerarquía, es decir, generar subclases de la clase que tenía los tipos que generaban el condicional. Por cada variante, crear un método en cada subclase que redefina el de la superclase. Copiar al método de cada subclase la parte del condicional correspondiente.

**Simplificación en la invocación de métodos**

Sirven para mejorar la interfaz de una clase.

1. **Rename Method**

Motivación: tener un código legible y autoexplicativo.

Mecanismo: cambiar el nombre actual del método a un nombre que explique más claramente para qué sirve el mismo.

1. **Preserve Whole Object**

Motivación: evitar pasar muchos parámetros, no se sabe cuando se van a requerir más datos del objeto en cuestión que brinda los valores.

Mecanismo: pasar el objeto entero del cual se piden los valores para enviar por parámetro, y que en el método que lo usa le pida los valores correspondientes.

1. **Introduce Parameter Object**
2. **Parameterize Method**

**Manipulación de la generalización**

Ayudan a mejorar las jerarquías de clases.

1. **Pull Up / Down Field**

Igual que el de abajo pero con field.

1. **Pull Up / Down Method**

Si los métodos en subclases son iguales → subir directamente Si los métodos en subclases no son iguales → parametrizar primero.

Si el método llama a otro que no está en la superclase, declararlo como abstracto en la superclase, esto mismo se puede aplicar con los field, es decir, subirlo arriba(pull up/down field) o self encapsulate field, es decir, crear un método en la subclase para acceder el field y luego hacer un método abstracto para el getter/setter en la superclase.

1. **Pull Up Constructor Body**

Subir arriba un constructor.

1. **Extract Subclass / Superclass**
2. **Collapse Hierarchy**
3. **Replace Inheritance with Delegation**
4. **Replace Delegation with Inheritance**

**The 2 hat**

Esto hace alusión a que no se puede desarrollar y refactorizar al mismo tiempo, dado que cuando se desarrolla se tiene una mentalidad creativa y se arreglan bugs mientras que al refactorizar se tiene que pensar de forma crítica.

Por esto se dice que se tienen 2 sombreros: adding function hat y refactoring hat.

Y no se pueden poner los dos sobreros a la vez.

**Automatización del refactoring**

Existen herramientas para hacer la refactorización más amena, sin embargo pueden ser algo restrictivas para preservar el comportamiento del programa (uso de precondiciones). Estas herramientas no pueden usarse si no existen tests para las funcionalidades a refactorizar.

Primera herramienta de refactoring: **Refactoring Browser (RB)**

Más adelante: herramienta **Code Critic** que detecta code smells.

Las herramientas solo chequean lo que sea posible desde el árbol de sintaxis y la tabla de símbolos. Además pueden ser demasiado conservativas (no realizan un refactoring si no pueden asegurar preservación de comportamiento) o asumir buenas técnicas de programación.

Algunas herramientas actuales:

* Smalltalk RB
* IntelliJ IDEA
* Sonar qube

**Patrones**

Existen varios tipos de patrones:

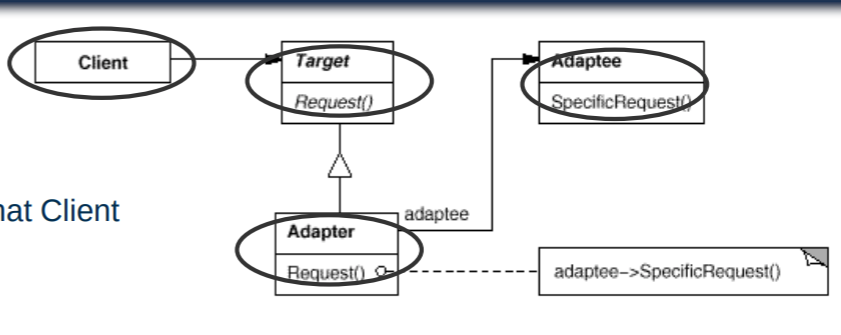
* Estructurales: tienen que ver con ampliar y formar objetos para generar una estructura más grande.
* Comportamiento: tienen que ver con los comportamientos de los objetos y posibles soluciones para mejorar su eficiencia, con sus algoritmos, delegación de responsabilidades y comunicación entre objetos.

Adapter (estructural) → wrapper

Se utiliza para cuando se tienen que clases que necesito usar en otra, sin romper el polimorfismo, en otras palabras, necesito adaptar los nombres de los métodos/campos para que la otra clase la pueda usar.

“Convertir” la **interfaz** de una clase en otra que el cliente espera. El Adapter permite que ciertas clases trabajen en conjunto cuando no podrían por tener interfaces **incompatibles**.

El adaptador puede ser un simple redireccionamiento como tener que realizar un procedimiento de transformación para la adaptación.



Participantes:

Target (Figure): define el dominio específico de la interfaz que el cliente usa.

Client (Editor): colabora con los objetos que tienen la interfaz de acuerdo al target.

Adaptee (3DFigure): tiene una interfaz que necesita de adaptación.

Adapter (3DAdapter) adapta el Adaptee de acuerdo a la interfaz de Target.

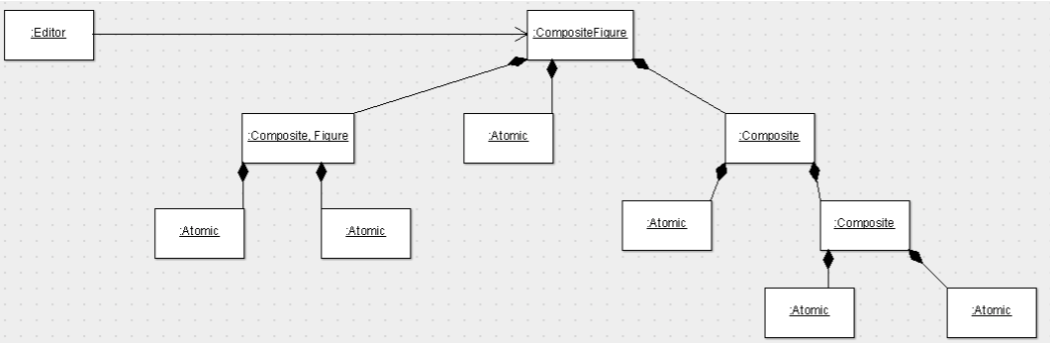
Composite (estructural)

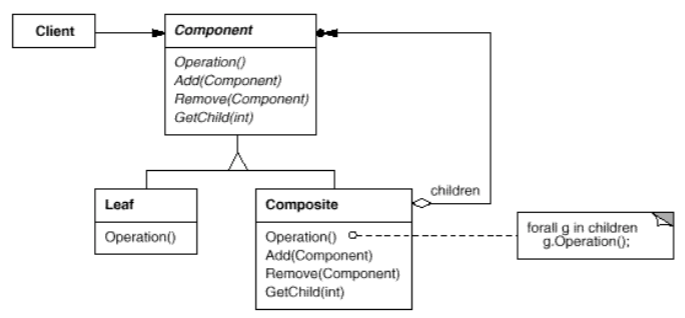
Sirve para generar composiciones de objetos que se puedan componer de sí mismos o de versiones atómicas de ellos.

Es decir, si se tiene una figura, una composición de esa figura podría estar formada por varias figuras simples o más figuras compuestas que dentro tengan simples o compuestas y así sucesivamente.

Esto ocasiona que se arme una especie árbol donde los métodos son recursivos entre ellos.

Como se ve en la imagen se tiene un árbol donde sus hojas son atómicas.





Intención:

Componer objetos en estructuras de árbol para representar **jerarquías parte-todo**. El Composite permite que los clientes traten a los objetos atómicos y a sus composiciones uniformemente.

Sirve para usarse cuando se necesitan jerarquías parte-todo o dicho de otra forma, cuando se quiere que la diferencia entre una composición o una figura simple sea transparente al cliente.

Participantes:

Leaf es la hoja, solo tiene las operaciones básicas y el composite tiene la misma interfaz que las hojas + las operaciones propias del composite como agregar figura, eliminar, etc.

Component (Figure):

**Declara la interfaz para los objetos de la composición**. Implementa comportamientos default para la interfaz común a todas las clases.

Desventaja: es difícil restringir el tipo de atómico que se quiere para una composición, es decir, cuando existen varios atómicos, y se tiene una composición que no puede tener cualquier tipo de atómico, requiere de una codificación más compleja para manejar este asunto.

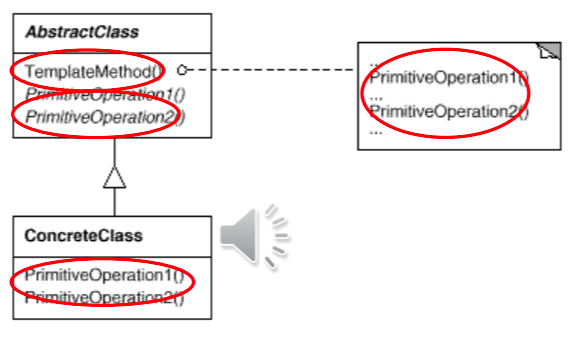
Template Method (comportamiento)

Se usa en ocasiones donde se tienen jerarquías que reescriben un mismo método, sin embargo, parte de este método se repite y solo por tener simples modificaciones deben reescribirse.

Por ejemplo, en una cuenta se verifican distintas cosas de la misma para poder realizar la extracción, pero dependiendo el tipo de cuenta se chequean cosas distintas.

Sin embargo, la finalidad siempre es extraer el dinero una vez pasado los chequeos.

En estos casos, se genera un método más general en la clase superior, para dejar que cada subclase reescriba los métodos que chequean y no todo el método extracción.



Entonces, en la clase superior, se define el método particular, y los métodos **hooks**, es decir, los primitivos como pueden ser los chequeos de las cuentas.

Siguiente a esto se definen en las subclases los métodos hooks que representen los distintos chequeos particulares y específicos de cada cuenta.

Un plus que se añade a la clase superior puede ser que los métodos hooks de la misma, no sean abstractos, si no que mantengan un resultado por default para aquellas subclases que no necesitan de dichos hooks.

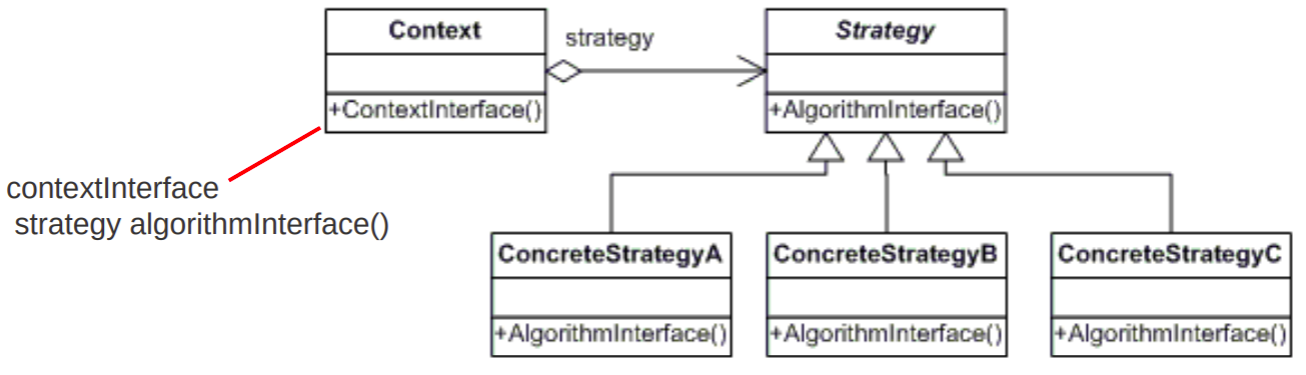
Intención: Definir el **esqueleto** de un algoritmo en un método, **difiriendo algunos pasos** a las **subclases**. El template method permite que las **subclases** **redefinan** ciertos **aspectos** de un algoritmo **sin cambiar** su **estructura**.

Aplicabilidad: *Para implementar las partes invariantes de un algoritmo una vez y dejar que las sub-clases implementen los aspectos que varían.*

Strategy (comportamiento)

Este algoritmo se usa para cuando se tiene distintos algoritmos para una misma clase dependiendo de la manera que se quiera ver dicha clase.

Es algo parecido al polimorfismo pero permite que se pueda cambiar la estrategia del algoritmo cuando se crea necesario. Se desacopla el algoritmo variante de la clase en cuestión.



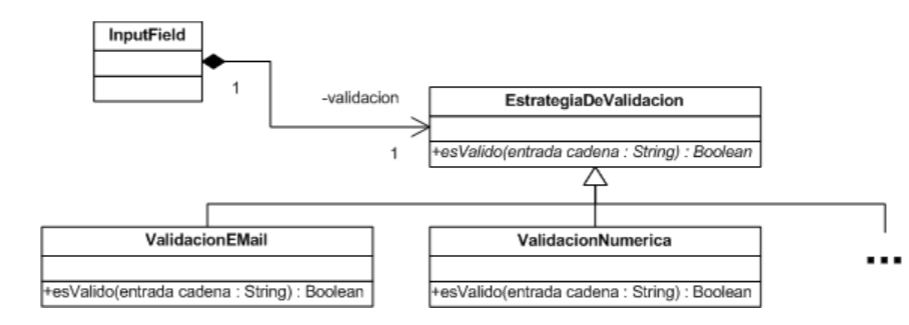
Una *desventaja* puede ser que existe un overhead de mensajes entre la clase(contexto) y la estrategia. Y que además complica el diseño cuando se podría solucionar con subclases o simplificando los condicionales.

Intención: Desacoplar un algoritmo del objeto que lo utiliza. Permitir cambiar el algoritmo que un objeto utiliza en forma dinámica. Brindar flexibilidad para agregar nuevos algoritmos que lleven a cabo una función determinada.

Aplicabilidad: *Existen muchos algoritmos* para llevar a cabo una tarea. No es deseable codificarlos todos en una clase y seleccionar cuál utilizar por medio de sentencias condicionales.

*Es necesario cambiar el algoritmo en forma dinámica.*

Ejemplo:



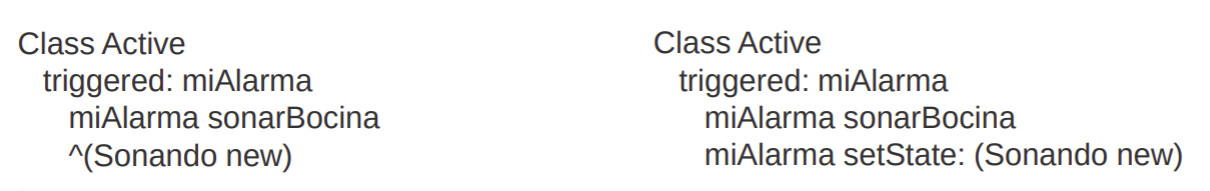
State (comportamiento)

Este patrón es muy parecido al strategy, el patrón state se utiliza cuando se tiene un objeto que cambia su comportamiento según un estado.

Por ejemplo, una alarma.

Entonces, para evitar condicionales anidados o muchas subclases de objeto, se desacopla el estado en un objeto aparte, lo que se llamaría state.

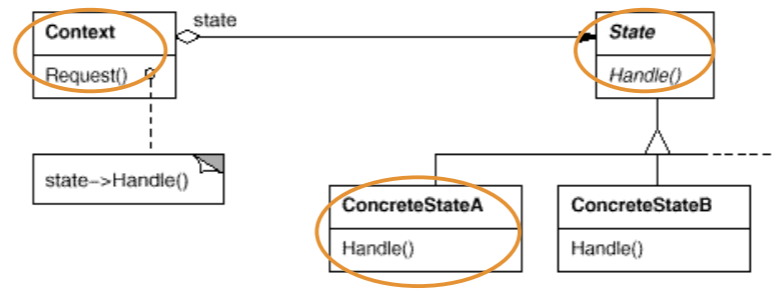
Además, el state a diferencia del strategy, debe conocer la alarma, dado que tiene que utilizar funciones de la misma para realizar su trabajo, y también, es el mismo estado el que sabe a que otro estado pasará la alarma una vez que hizo lo que debía.



(se puede establecer el estado de la alarma, con setState o retornar el nuevo estado para que ella misma lo haga)

Intención: **Modificar** el **comportamiento** de un objeto **cuando** su **estado interno** se **modifica**. Externamente parecería que la clase del objeto ha cambiado.

Aplicabilidad: El **comportamiento** de un objeto **depende** del **estado** en el que se encuentre. Los **métodos** tienen **sentencias condicionales** complejas que **dependen** del **estado**. Este estado se representa usualmente por **constantes** enumerativas y en muchas operaciones aparece el mismo condicional. El patrón State reemplaza el condicional por clases (es un uso inteligente del polimorfismo)



¿State o Strategy?

Si bien son parecidos, estos patrones tienen propósitos distintos.

El patrón State es útil para una clase que debe realizar **transiciones** entre estados fácilmente. El patrón Strategy es útil para permitir que una clase **delegue** la ejecución de un **algoritmo** a una instancia de una familia de estrategias.

Entonces, en el state se pueden conocer los distintos estados y entre ellos pueden mandarse mensajes, por otro lado, el strategy no conoce a los demás strategies y el encargado de cambiar la estrategia es el cliente, por lo tanto no es transparente como lo es el state.

Otra diferencia adicional es que cada State puede definir muchos mensajes, pero un Strategy suele tener un único mensaje público.

Wrapper: decorator (estructura)

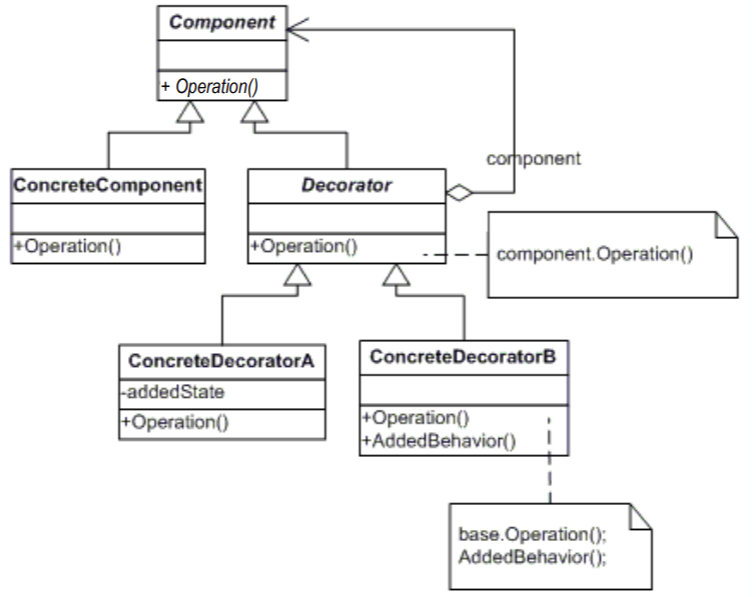
Este patrón se utiliza para cuando se quiere **agregar** o **quitar** comportamiento de forma dinámica sin que el objeto en cuestión se entere, es decir, de forma transparente. Si bien se podrían generar subclases, estas no son prácticas a la hora de quitar y agregar de forma dinámica.

En el ejemplo de los filtros, uno podría agregar o quitar filtros cuando uno quisiera, si bien se podría generar una solución tipo composite, se estaría delegando todo el protocolo al objeto de contexto y no sería transparente.

Por lo tanto se tendría una jerarquía donde se tiene una interfaz general de donde sale el objeto principal y además los decoradores.

¿Porque los decoradores son subclases en esta jerarquía? Es necesario dado que los decoradores son los que van a contener a otros decoradores o al objeto principal, por ende tiene que tener el mismo interfaz para mantener la comunicación entre ellos.

La estructura sería esta:



*Desventaja*: Mayor cantidad de objetos, complejo para depurar.

Diferencia con el adapter:

Ambos patrones “decoran” el objeto para cambiarlo •

*Decorator* ***preserva la interfaz*** *del objeto para el cliente. Adapter* ***convierte la interfaz*** *del objeto para el cliente*.

Los decorators se suelen **anidar**, los adapter no se anidan directamente.

Diferencia con el composite:

Si bien su estructura se parece al composite no es igual, porque la clase “concreta” no es el decorador, como lo era el compuesto en el composite, si no que es un envoltorio que se aplica sobre la clase concreta. Además, si bien pueden haber varios decoradores aplicados, no están todos a la misma altura, es decir, los decoradores se aplican unos sobre otros, como con los filtros, por esto no pueden ser una colección como en el composite. En resumen, las diferencias más importantes radican en que su **propósito** **no es el mismo** y además, el composite genera un árbol y el decorador una lista, por ende, el **parecido en la estructura** radica en que se quiere **generar** un **anidamiento**.

Diferencia con el strategy:

El **Strategy** cambia el algoritmo por **dentro** del objeto, el **Decorator** lo hace por **fuera**.

Wrapper: proxy (estructura)

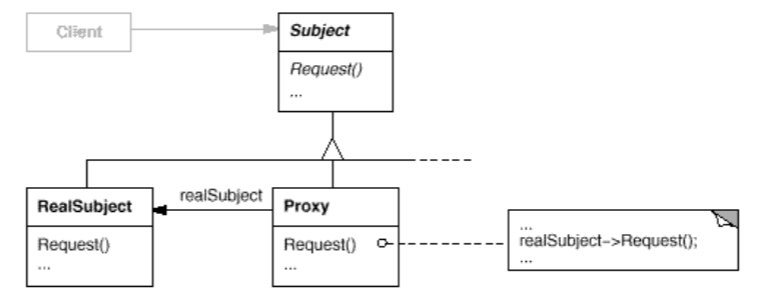
Se utiliza cuando se quiere ya sea un intermediario para controlar el acceso por ejemplo derivar si el formulario se utiliza para edición o para un alta.

O para crear un objeto “falso” que ocupe el lugar del objeto real mientras se evalúa o procesa este último.

Este objeto falso, llamado proxy va a tener la misma interfaz del objeto original, sin embargo va a tener funciones que no va a poder manejar y deberá buscar la imagen real.

Intención: proporcionar un **intermediario** de un objeto para **controlar** su **acceso**.

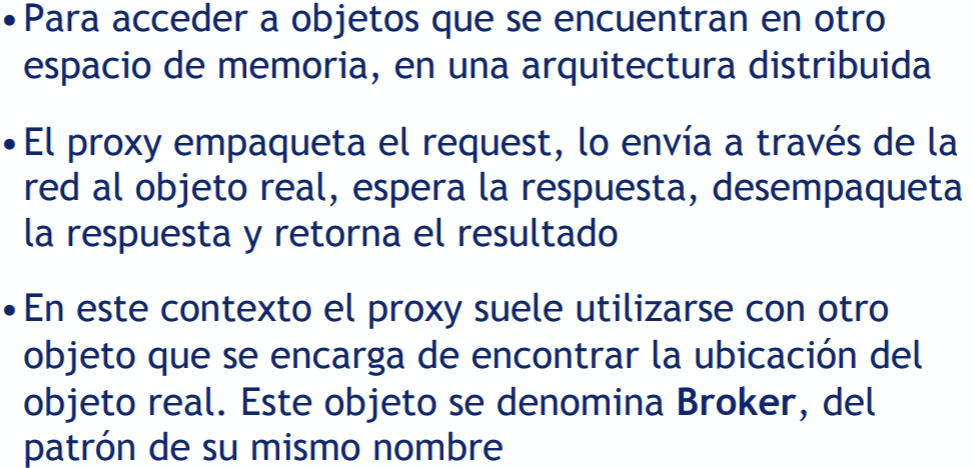
Aplicabilidad: cuando se **necesita** una **referencia** a un objeto más **flexible** o sofisticada.



Si se observa, el proxy no pasa por la superclase, si no que tiene una relación directa con el objeto real, esto es porque no se anidan los proxys pero un proxy si tiene la misma interfaz que el objeto real.

Existen varios tipos de proxys:

* Virtual Proxy: Demorar la construcción de un objeto hasta que sea realmente necesario. (+ eficiencia)
* Protection Proxy: Restringir el acceso a un objeto por seguridad (por ejemplo ante spam, aunque en este caso podría ser un protection y un virtual proxy a la vez).
* Remote Proxy: para implementar objetos **distribuidos**.



Se combina con otro patrón denominado broker(sabe dónde está cada objeto real).

Un proxy se puede implementar de otra manera que no necesite la interfaz real. La razón de plantear esto es que por ejemplo, si se tiene una bd y se quieren generar proxys, habría que tener un tipo de proxy por cada objeto existente que vaya a guardarse en la misma, esto es demasiado costoso.

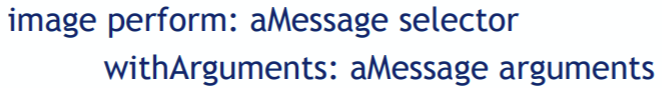
Por ello se tiene otra forma de implementar proxys: reflexion.

*Implementación de proxy con* ***reflexión****:*

En smalltalk todo se trata como un objeto, inclusive los mensajes. Cuando no se encuentra un metodo en la clase que pueda responder al mensaje se busca en la jerarquía superior, si esta búsqueda llega a la clase object y no existe entonces ahí aparece el metodo doesNotUnderstand. Este recibe un objeto mensaje el cual contiene un selector (nombre) y varios argumentos.

Cuando se tiene un proxy que no mantiene una interfaz de acuerdo al objeto real, se puede reescribir el metodo doesNotUnderstand y realizar todas las validaciones correspondientes para después derivar el mensaje al objeto real que si sabra responderlo. De esta manera se puede tener un único proxy para todos los objetos que puedan guardarse en una bd.

Smalltalk tiene un metodo perform que puede recibir un selector y argumentos para enviarselos a un objeto. Este es un metodo de object, por lo tanto lo heredan todos.



**Reflexión**

La reflexión es la capacidad que tiene un programa de observarse y modificarse a sí mismo de forma dinámica.

Es aquel que puede **observar** su propia **ejecución** (**introspección**) e incluso cambiar la manera en que se ejecuta (**intercesión**, ej: cambiar el metodo doesNotUnderstand). Para poder realizar esto se requiere poder expresar y manipular el estado de la ejecución como datos, es decir, que la misma ejecución del programa pueda objetificarse, como en el caso de los mensajes que si bien son funcionalidades de ejecución del programa, a su vez son objetos. Esto último se denomina **reificación**.

**Refactoring to patterns**

Poca ingeniería (under-engineering) significa producir software con un diseño pobre. Pero por otro lado, la sobre-ingeniería (tener software más sofisticado de lo que debería ser) es tan peligrosa como la poca ingeniería, porque a veces complica el mantenimiento para cosas que son más simples sin el patrón.

Por eso justamente, el refactoring está para aplicar patrones cuando se crea necesario, una vez que se ve que sea necesario.

Existen refactorings que son guías para poder pasar un modelo a un patrón conocido.

“Los patrones son tentadores para no quedarnos envueltos y arrastrar un mal diseño. También nos pueden llevar al otro extremo. Por esto es muy importante conocer las consecuencias tanto positivas como negativas de un patrón. Introducimos patrones sólo cuando se **necesitan**”.

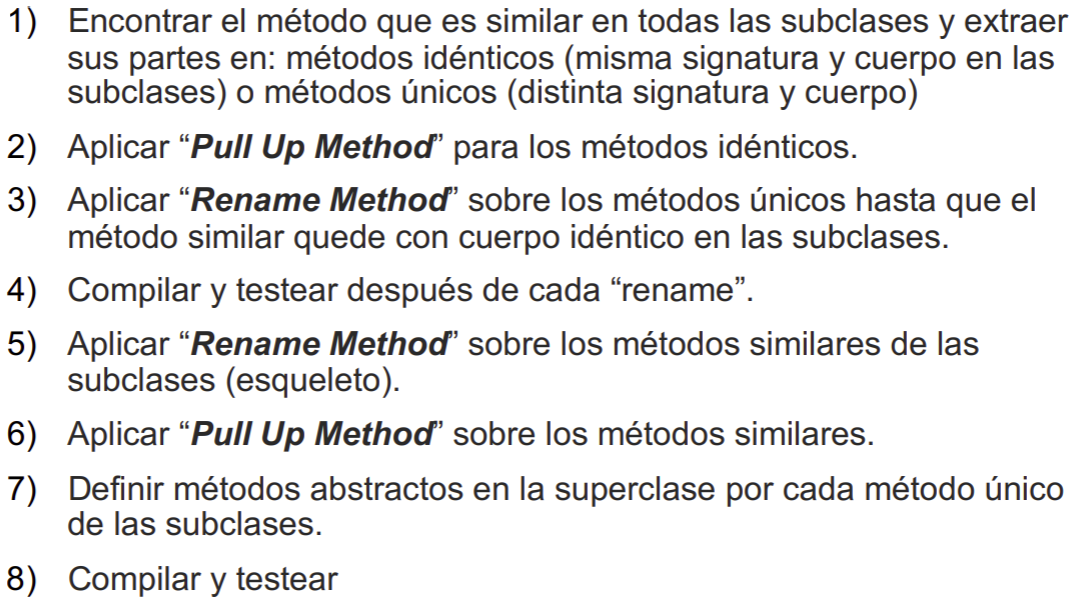
**Patrones**

*Template method*

El template method ayuda a eliminar código repetido en métodos similares de las subclases de una jerarquía.

Mecánica:

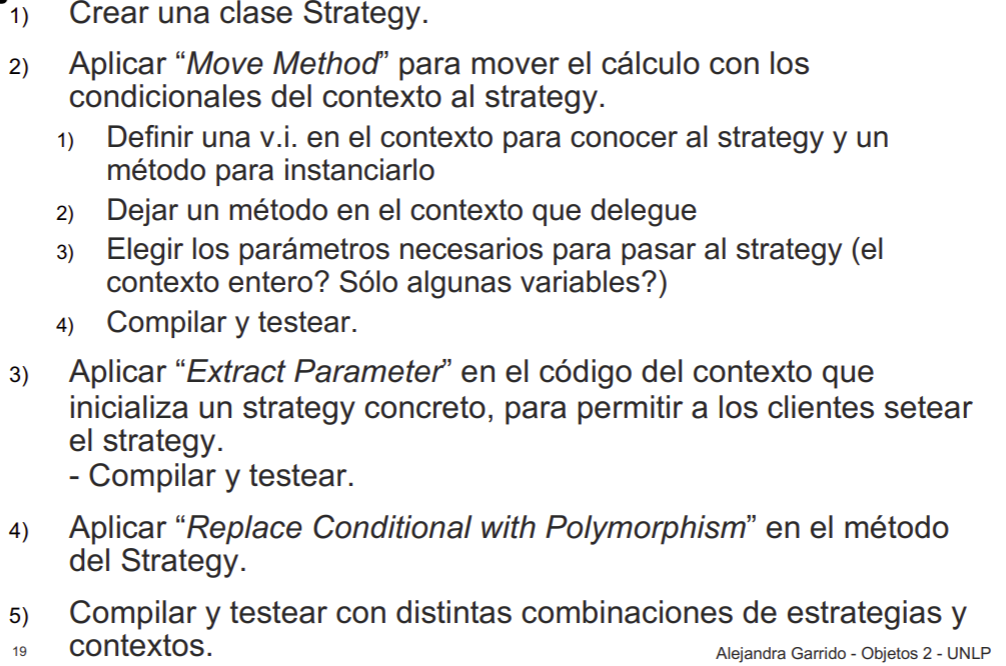
1. Encontrar el método que es similar en todas las subclases y extraer sus partes en: métodos idénticos (misma signatura y cuerpo en las subclases) o métodos únicos (distinta signatura y cuerpo)
2. Hacer un pull up method y rename method de los métodos idénticos.
3. Rename method de las clases únicas.
4. Definir métodos abstractos en la superclase por cada método único de las subclases.
5. Compilar y testear



Desventaja: **Complica** el diseño cuando las **subclases** **deben** implementar **muchos** **métodos** para **sustanciar** el **algoritmo**.

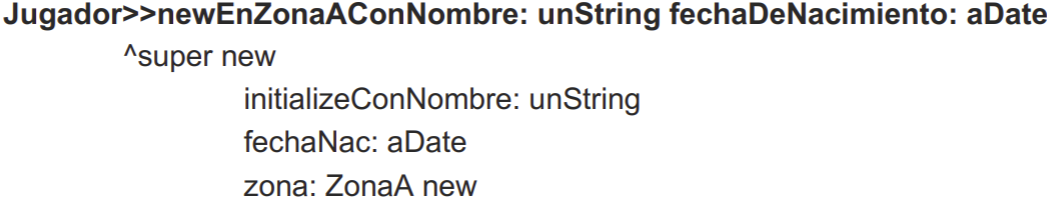
*Strategy*

Existe lógica condicional en un método que controla qué variante ejecutar entre distintas posibles. Crear un Strategy para cada variante y hacer que el método original delegue el cálculo a la instancia de Strategy.



→→ Extract Parameter: es agregar un parámetro al constructor que permita setear el valor que el programador quiera.

En el caso de que no haya demasiadas subclases de strategy, se puede usar el refactoring llamado *Encapsulate Classes with Factory*, que básicamente lo que hace es generar un constructor con el nombre de la estrategia, el cual setee automáticamente una instancia de la estrategia que indica el nombre.



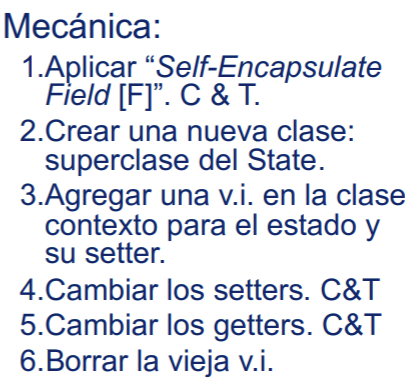
Se debe aclarar que esto es un “factory” o “creation method”, pero no es el patrón Factory Method.

*State*

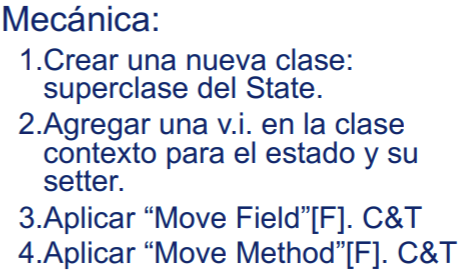
Cuando los condicionales que reflejan las transiciones entre estados se vuelven complejos es necesario usar el patrón State.

Por ende se sigue una mecánica teniendo en cuenta dos posibilidades:

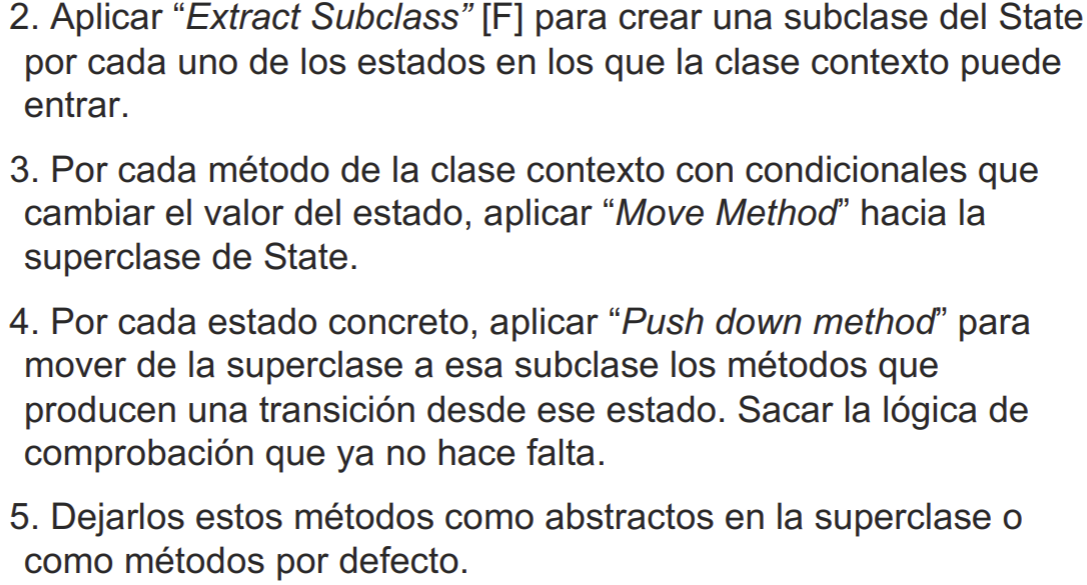
* Existe una única variable que maneja el estado:



* Existen varias variables que manejan el estado:



Luego de esto, la mecánica es igual para ambos casos:

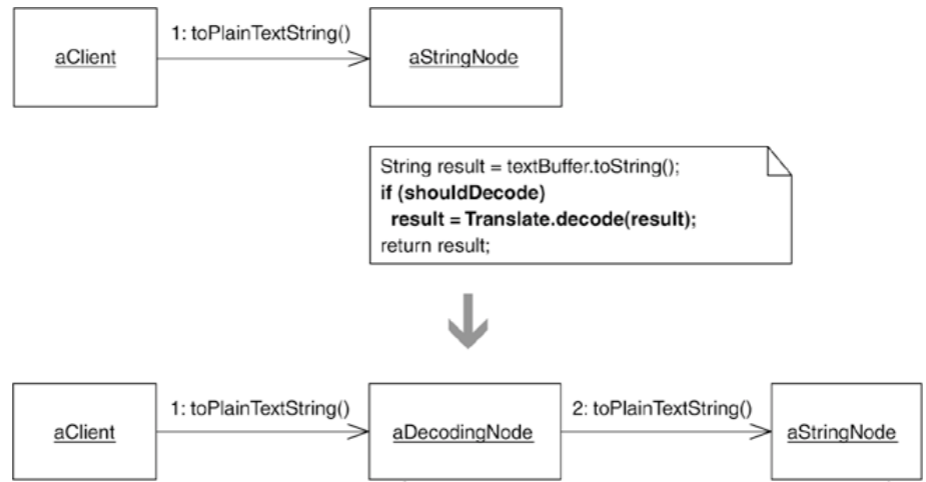


*Desventaja*: Complica el diseño cuando la lógica de transición de estados ya es difícil de seguir.

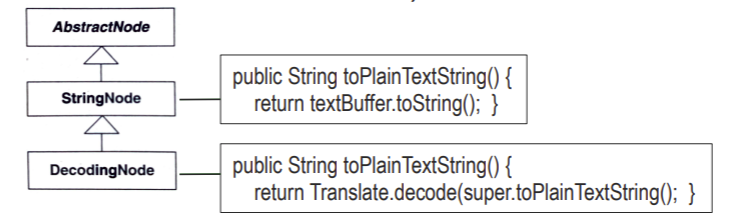
*Decorator (Move Embellishment to Decorator)*

Sirve para quitarse condicionales de encima que podrían ser decoradores.

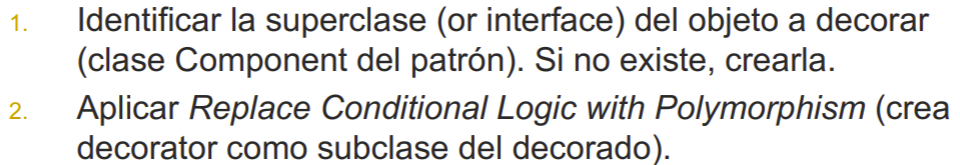
Por ejemplo:



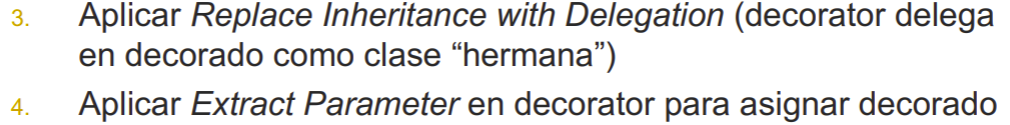
Existen algunos casos como el que se ve acá, que podrían aplicar el mecanismo de conversión al patrón por la mitad y sería suficiente.



Como se ve en la imagen, el decorador no está delegando toPlain a un StringNode, si no que este es su superclase. Para esta situación esto es suficiente porque arregla el problema, por ende no es necesario seguir con la transformación que sugiere el patrón, el cual tendría al decorador cómo subclase de una interfaz y luego se anidaria etc, etc.



Hasta el punto 2 bastaría para el caso anterior.



**Refactoring de UX**

**Fragile Test** es un test smell que proviene de un test que falla en compilar o ejecutarse cuando el SUT cambia en formas que no afectan la parte que el test ejercita (es decir, hay acoplamiento entre el test y el SUT más allá de la funcionalidad a ejercitar)

Una de las causas de Fragile Test es “**sensibilidad a la interfaz**”, o “sensibilidad al protocolo” del SUT.

Refactorings de Fragile Test:

*Test Utility Method*: permite encapsular la dependencia innecesaria con el API del SUT en un método, por ejemplo, de creación. ¿Cuándo es innecesaria? Cuando se refiere a un método del SUT que no es el que se está testeando.

*Creation Method*: es un tipo de Test Utility Method que oculta la mecánica de crear objetos ready-to-use atrás de un método con nombre adecuado (Intention-Revealing Names).

**Patrones de IU web**

Patrones de diseño de interfaz de usuario (IU) son soluciones recurrentes que resuelven problemas comunes de diseño de aplicaciones web.

Es difícil diseñar la UI con patrones, por esto se requiere poder **adaptarse**, **innovar** y **aprender del feedback**.

Propósito: mejorar la calidad externa de una aplicación web, es decir, la usabilidad, accesibilidad UX, preservando la funcionalidad

Alcance: navegación ,presentación, interacción del usuario

Malos olores de UX

* Scarce Search results: no se autocompletan las búsquedas y por eso no se encuentran resultados.

Solución: Add autocomplete.

* Late Validation: No se muestran las validaciones antes de enviar el formulario, ni se expresa cuales son los campos obligatorios.

Solución: Add Inline Validation

* Unresponsive Elements: los elementos no son clickeables, no redirigen cuando se necesita poder observar un atributo más específicamente.

Solucion: Turn Attribute into Link

*Client-Side Web Refactorings (CSWR)*: Son Refactorings que solucionan malos olores de UX aplicando cambios del lado del cliente, sobre el DOM de las páginas web.

**Reuso de framework**

Existen varios tipos de cosas que se pueden reusar en la programación, como por ejemplo: aplicaciones completas(Wordpress), componentes y servicios (APIs), Funciones y estructuras de datos, diseños o estrategias de diseño (patrones) o inclusive ideas.

Las razones de reutilizar pueden ser varias:

* Aumentar la productividad
* Aumentar la calidad

Sin embargo existen algunos problemas:

* Problemas de **mantenimiento** si no tenemos **control** de los componentes que reusamos.
* Algunos programadores prefieren hacer todo ellos mismos (**confianza** en lo **propio**, búsqueda de **desafíos**)
* Hacer software reusable es más difícil y costoso (paga a la larga y requiere procesos especiales)
* Encontrar, aprender a usar y adaptar componentes reusables requiere **esfuerzo adicional**

Pueden haber varios tipos de clasificaciones de códigos a reutilizar, como pueden ser frameworks, librerías o inclusive patrones de diseño.

Librerías de clases

Son aquellas utilidades que resuelven problemas comunes que se presentan en todas las aplicaciones, como colecciones, fechas, funciones aritméticas,etc.

En este caso, todas las librerías son independientes del contexto y pueden ser usadas en cualquier aplicación, por esto podemos decir que *nuestra aplicación controla a la librería*.

Framework orientado a objetos

Un framework, como la palabra lo dice, es un marco de trabajo sobre el cual pulir y crear nuestra aplicación. Es una aplicación **semi-completa** que se puede **reutilizar**, es un **conjunto** de **clases concretas y abstractas**, **relacionadas** para proveer una **arquitectura reusable** para una familia de aplicaciones relacionadas.

Las **clases** en el **framework** se **relacionan** (herencia, conocimiento, envío de mensajes) de manera que **resuelven la mayor parte** del problema en cuestión – **conforman un todo**.

A diferencia de las librerías, el código del framework es el que controla al nuestro, este cambio de quien controla a quién se lo conoce como **inversión de control**.

Hay dos conceptos clave en un frameworks: Frozenspot vs Hotspots

Frozenspot: son aquellas **partes** **estáticas** que forman parte de un framework las cuales **no** se pueden **cambiar** o alterar. Son los desarrolladores del framework los que imponen las reglas y todas las **apps** que **usen** dicho **framework** **comparten** esta parte.

Hotspots: son las partes que un desarrollador que utilice el framework puede modificar y crear, son **puntos de extensión** que nos permiten introducir variantes y así construir aplicaciones diferentes.

Los puntos de extensión pueden implementarse en base a **herencia** o en base a **composición**.

A los frameworks que utilizan **herencia** en sus puntos de extensión, es decir, aquellos donde se necesita entender el código interno del mismo para poder subclasificar las clases, les llamamos de **Caja Blanca** (Whitebox).

Por otro lado, los que utilizan composición (ofrecen componentes a elección que solo deben configurarse) les llamamos de **Caja Negra** (blackbox).

**Inversión de control**

La inversión de control es la característica principal de la **arquitectura run-time** de un **framework**.

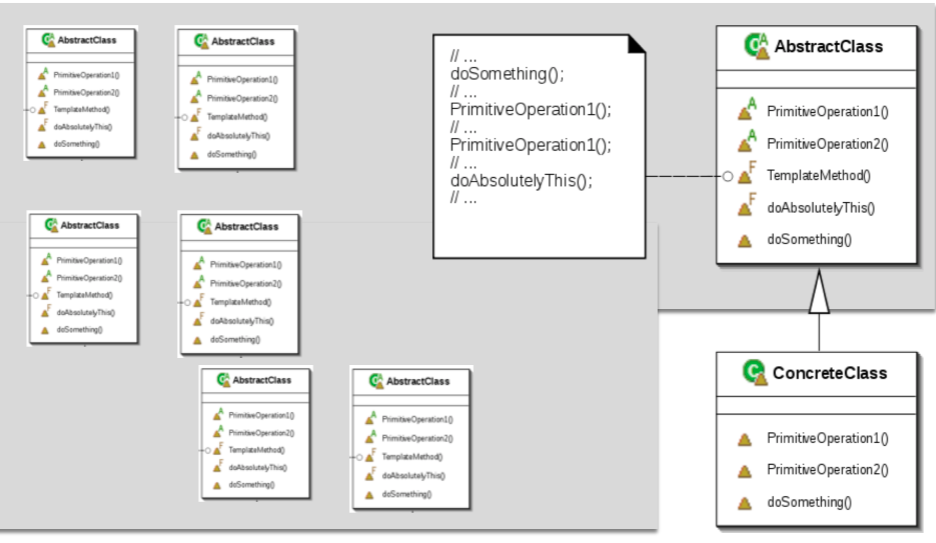
Esta característica permite que los **pasos** **canónicos** de procesamiento de la aplicación (comunes a todas las instancias) sean **especializados** por objetos tipo **manejadores de eventos** a los que **invoca** el framework como parte de su mecanismo reactivo de despacho.

**Template method en frameworks**

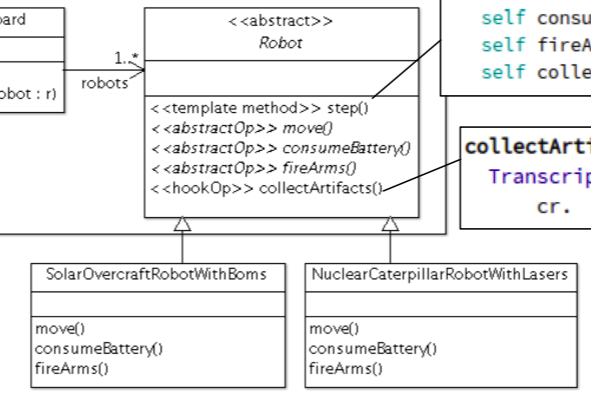
Los framework tienen un patrón conocido que se utiliza en el frozenspot, el cual es el template method.

Este proporciona una plantilla para que el desarrollador pueda adaptar su aplicación sin irse de ese esquema o interfaz que le brinda el frozenspot.

(Imagen: Gris es el frozenspot, la parte blanca el hotspot)



Entonces, para extender un punto de extensión, se pueden usar dos métodos: herencia o composición.

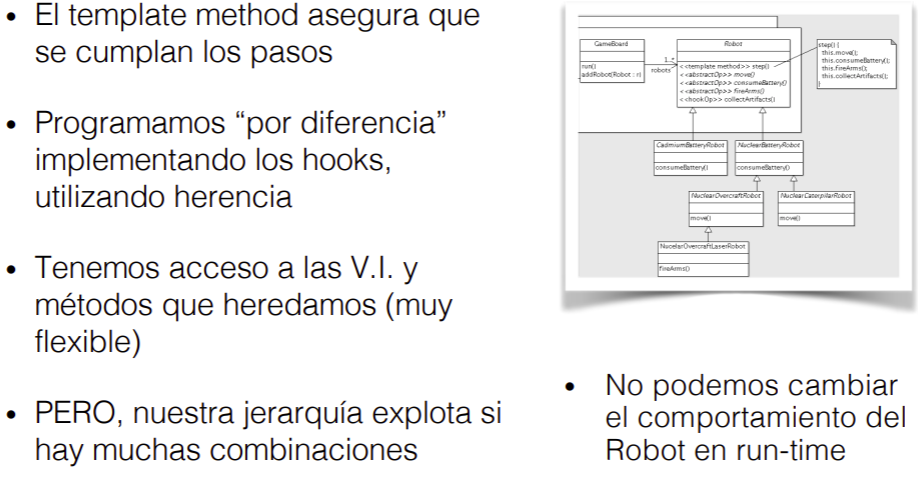
La herencia consta de generar subclases que hereden de la plantilla y redefinen los métodos que la misma provee como abstractos.

Por ejemplo, la clase robot vendría a ser la plantilla, mientras que abajo se van definiendo los métodos que especializan al robot, como la manera en que se mueven, que disparan, que batería usa, etc.

Sin embargo, existe un problema el cual radica en que pueden existir muchas combinaciones entre los tipos de robot.

Por ejemplo, pueden haber dos robots que se mueven igual y disparen lo mismo pero su batería sea distinta, por lo cual habría duplicado de código.

Si bien esto último podría resolverse con una jerarquía que junte lo común y subclasifique lo diferente, provocaría que haya una cantidad enorme de jerarquías y subclasifcaciones hasta el punto de hacerlo demasiado extenso y difícil de seguir.

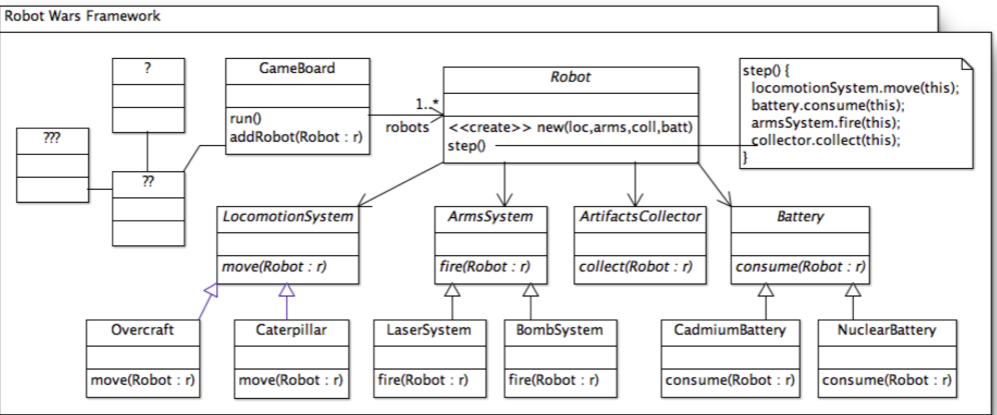


En run-time significa que no se puede cambiar el tipo de robot en ejecución.

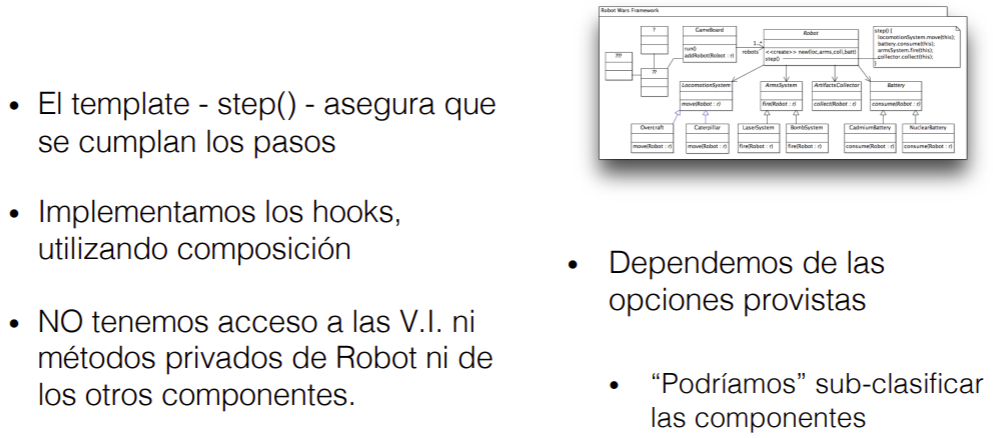
Entonces existe otro método para solucionar esto el cual es la composición.

En la composición se van a generar especie de strategies por cada funcionalidad presente en la clase Robot.

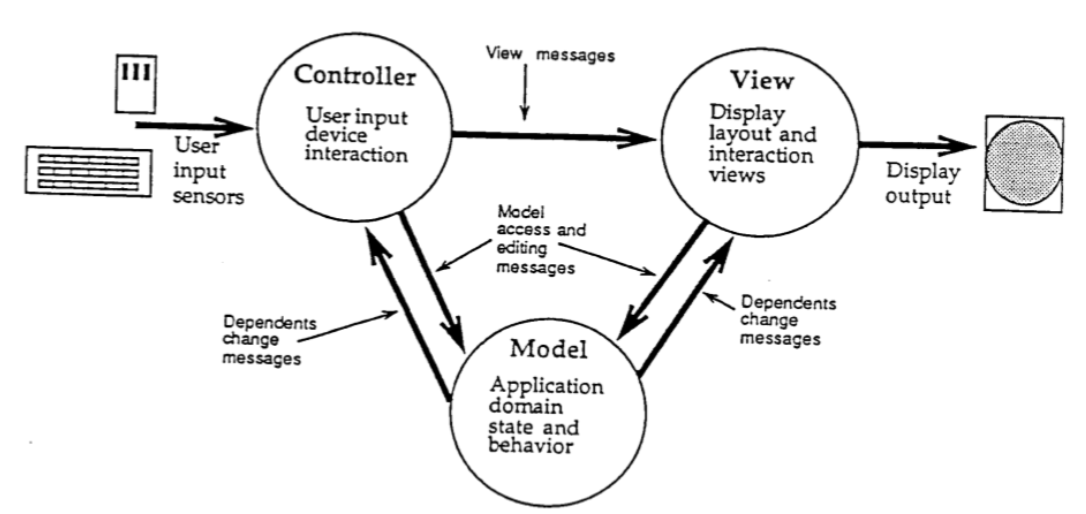
Entonces por cada tipo de batería se crearian subclases pero que únicamente tengan que ver con la batería.



Sin embargo, en este caso se tiene la desventaja de que no hay acceso desde un tipo de compuesto a otro, por ende cada vez que se usan los mismos deben enviarse a sí mismos para que la clase robot efectúe cambios en los demás componentes si es necesario.



**Paradigma Model-View-Controller (MVC)**



Hacer widgets reutilizables implica:

* Programar lo que se ve
* Programar lo que se hace
* Programar los objetos subyacentes

Se sabe que es una buena práctica separar el código de dominio del código de presentación para evitar acoplarlos entre ellos y que de esa manera sea fácil de mantener.

Este paradigma cuenta con tres componentes:

* View: es el que se encarga de mostrar visualmente al modelo. Este componente conoce al modelo dado que tiene que pedirle lo que debe mostrar.
* Controller: es el encargado de responder ante los eventos ocasionados por el usuario, y determinar qué debe hacer el view y el modelo ante estos.

Hay un controlador por cada view.

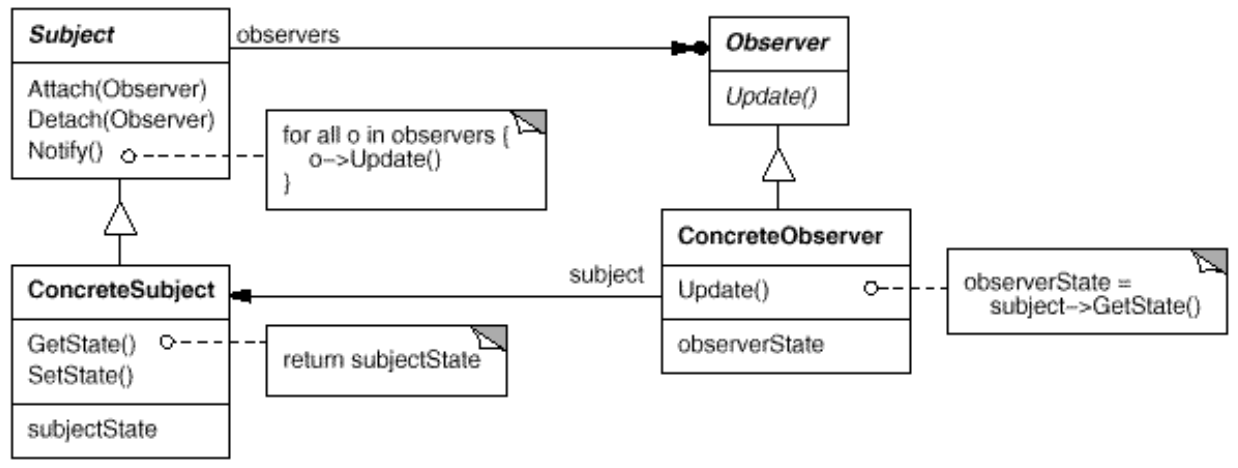
* Model: Es el que lleva el código del dominio del widget y mantiene el estado. Este no debe estar acoplado al resto de componentes.

Sin embargo existe un problema, ¿cómo se mantiene actualizado el view cuando algo cambia en el modelo?

Para eso se va a necesitar de un modelo que avise ante un cambio. Es entonces cuando surge un patrón denominado “Observer”.

**Patrón observer**

Intención: Definir una **relación** **uno-a-mucho**s entre objetos de forma que **cuando** un objeto **cambia** su **estado**, **todos** sus dependientes son **notificados** y actualizados **automáticamente**.



Sujeto (Subject)

El sujeto conoce a los observadores que lo están observando pero nunca se entera a ciencia cierta cuántos hay.

Provee una interfaz (mensajes) para registrar y desregistrar a los observadores.

Sujeto Concreto (ConcreteSubject)

Es quien almacena el estado que les interesa a los ObservadoresConcretos. Este es quien notificará a los observadores cuando su estado cambia.

Observador (Observer)

Define una interfaz que utilizará los observadores concretos y el subject para comunicar que se produjo un cambio.

ObservadorConcreto (ConcreteObserver)

Es el observador concreto que mantiene una referencia a un SujetoConcreto.

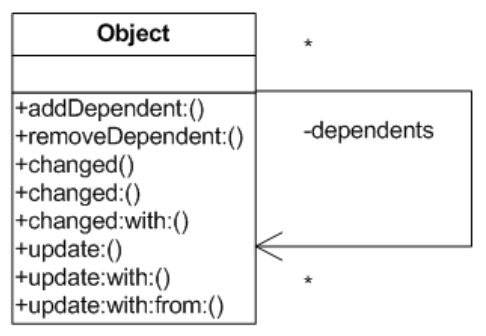
Almacena estado que debe mantenerse consistente con el estado del SujetoConcreto.

**Implementa** la interfaz de actualización definida en la clase **Observador** para mantener su estado **consistente** con el de su SujetoConcreto.

Si bien el sujeto conoce a los observadores, no sabe cuántos son ni qué tipo de objetos son y tampoco le interesa saberlo. Lo único que **exige** para comunicar su actualización es que el **observador** **implemente** la interfaz **Observer** que contiene el mensaje **update** que el sujeto conoce.

Entonces existe una **colaboración** entre ellos, un bucle entre que el modelo avisa, el view se entera y le pide al modelo sus datos actualizados.

Smalltalk provee una implementación del patrón observer que sirve y entienden todos los objetos que heredan de object (changed = notify; addDependent = attach).



**Notificación con parámetros**

La notificación de cambio puede ir acompañada de argumentos, por ejemplo un indicador del aspecto que cambió (un string).

El observador recibe el mensaje update() con los argumentos pasados por el sujeto, y lo utiliza para determinar la magnitud del cambio.

Algunas implementaciones utilizan objetos complejos como argumentos. Esos objetos reciben el nombre general de anuncios (**Announcements**) y permiten implementar variantes más ricas del patrón observador.

Sin embargo esto puede generar acoplamiento si no se trata con cuidado.

Aplicabilidad:

* Cuando una **abstracción** tiene **dos** **aspectos**, uno **dependiente** del otro. Encapsular estos aspectos en objetos separados permite que los mismos varíen y sean reutilizados independientemente.
* Cuando un **cambio** en un objeto requiere que **otros** objetos **cambien**, y usted no quiere saber **cuántos** otros objetos necesitan cambiar.
* Cuando un objeto debe ser capaz de **notificar** a otros sin hacer conjeturas en relación a **quienes** son esos objetos. En otras palabras, **no** se quiere a estos objetos **fuertemente** **acoplados**.

Conclusiones:

* El patrón Observador permite **variar** sujetos y observadores independientemente. Se puede **reusar** a los **sujetos** **sin** **reusar** los **observadores** y viceversa. Se pueden agregar nuevos observadores sin modificar el sujeto ni otros observadores.
* **Acoplamiento abstracto** entre Sujeto y Observador: Todo lo que un sujeto sabe es que tiene una lista de observadores, cada uno de los cuales implementa la interfaz de la clase abstracta Observador. El sujeto no conoce la clase concreta de ningún observador. Por tanto el acoplamiento entre observador y sujeto es **abstracto** y **mínimo**.
* Soporte para **comunicación** **broadcast** (uno a muchos): a diferencia que en un envío de mensaje tradicional, el **sujeto** **no necesita indicar destinatario** (receptor). Las **notificaciones** son **distribuidas** automáticamente a **todos** los objetos interesados que se suscribieron.