# RESUMEN OBJETOS 2

## Closures: Clausuras Léxicas

- Los closures nos permiten diferir la evaluación de expresiones.
- Una clausura es una expresión que se define en el contexto de otra expresión.
- Situaciones frecuentes comunes donde utilizo clausuras:
    - Para indicar como continuar cuando esté listo el resultado de una tarea que se ejecuta de manera asincrónica (Callbacks).
    - Cuando quiero tener funciones que toman otras funciones como parámetro o que retornan funciones que luego voy a utilizar (por ejemplo: iteración).
    - En general, cuando quiero que sea posible cambiar (en parte) el comportamiento de un objeto, sin necesidad de su clasificar o modificar el código existente.
        - Muy útil en la construcción de librerías o framework.
- En la clausura se puede hacer referencia a elementos definidos en su contexto (por ejemplo variables).
- La pseudovariable "self" corresponde al contexto en el que define la clausura.
- La clausura se "lleva" las variables del contexto en el que fue definida. Si cambian "afuera" afecta a la clausura. Si se modifican en la clausura, tiene efecto "afuera".

## BlockClosure: Bloques

- En Smalltalk, los bloques (por ejemplo, [Transcript show: 'hola']) son clausuras.
- Una expresión entre corchetes crea un bloque.
- Puede hacer referencia a cualquier variable disponible en el contexto en que se define (por ejemplo, variables de instancia, variables temporales de un método, parámetros de un método).
- Puedo asignarlo a variables y pasarlo como parámetro.
- Puedo hacer referencia a parámetros que deberá recibí al momento de ser evaluado.
- Se evalúa enviando variantes del mensaje #Value.
- Cuidados de los bloques:
    - Los bloques son muy potentes, pero no debemos abusar.
    - Complican la depuración y hacer los programas más difíciles de entender.
    - Una vez creados, no se pueden modificar (hay que volver a crearlos).
    - No utilidad bloques cuando lo mismo puedo conseguirlo sin ellos.

## SUnit – Unit Testing

- Contexto:
    - Nos interesa incrementar la calidad del software: funcionalidad correcta (lo que espera el cliente) y que funcione correctamente (sin errores).
    - Testing se puede realizar a distintos niveles: Test de aceptación, de integridad y de unidad.
    - Testing efectivo asume la presencia de un framework de unit-testing (como los de la familia xUnit) que permite automatizar los test.
        - Volver a ejecutar una y otra vez los test creados.
        - Visualizar el resultado fácilmente.
- Framework xUnit:
    - La primera letra identifica el lenguaje: SUnit, JUnit, CppUnit, NUnit, PyUnit, etc.
    - La primera herramienta de testing automático fue SUnit, escrito por Kent Beck para Smalltalk.
    - Por su simplicidad y funcionalidad, se llevó prácticamente a todos los lenguajes de programación, y es open source.
- Test de Unidad (xUnit):
    - Testeo de la mínima unidad de ejecución.
    - En POO, la mínima unidad es un método.
    - Objetivo: aislar cada arte de un programa y mostrar que funcione correctamente.
    - Cada test confirma que un método produce el output esperado ante un input conocido.
    - Es como un contrato escrito de lo que esa unidad tiene que hacer.
    - Parte de un test de unidad:
        - Fase 1: Fixture Set Up
            - Preparar todo lo necesario para testear el comportamiento del SUT (unidad).

        - Fase 2: Exercise
            - Interactuar con el SUT para ejercitar el comportamiento que se intenta verificar.
        - Fase 3: Check
            - Comprobar si los resultados obtenidos son los esperados, es decir si el test tuvo éxito o fallo.
        - Fase 4: Tear Down
            - Limpiar los objetos creados para y durante la ejecución del test.
- SUnit – Pasos a seguir:
    - Crear clase TestCase que tenemos que subclasificar.
    - Redefinir los métodos SetUp (donde se crea el Fixture), TearDown (donde se borran los objetos creados), y escribir métodos de testing que empiecen con la palabra "test".
    - Ejecución: 1-SetUp. 2-Metodo Test. 3-TearDown.
- ¿Qué valores testear?
    - Escribir casos de testing es deseable pero es costo.
    - Testear todos los valores no es práctico.
    - Se busca encontrar casos importantes: aquellos donde es más probable que comentan errores.
        - Particiones equivalentes.
        - Valores de borde.
- Particiones equivalentes
    - Tratar conjuntos de datos como el mismo (si un test pasa, otros similares pasaran).
    - Para rangos, elegir un valor en el rango, y un valor fuera de cada extremo del rango.
    - Para conjuntos, elegir uno en el conjunto y uno fuera del conjunto.
- Valores de borde
    - La mayoría de los errores ocurren en los bordes o límites entre conjuntos.
    - Los "valores de borde" complementa a "particiones equivalentes".
- ¿Cómo? ¿Cuándo? ¿Por qué testear?
    - Saber porque se testea algo y a qué nivel debe testearse.
    - El objetivo de testear es encontrar bugs.
    - Se puede aplicar a cualquier artefacto del desarrollo.
    - Se debe testear temprano y frecuentemente.
    - Testear tanto como sea el riesgo del artefacto.
    - Un test vale más que la opinión de muchos.

**Introducción a Refactoring**

- Algunas Leyes de Lehman: (ocurren en el mantenimiento de software).
    - Continuing Change: los sistemas deben adaptarse continuamente o se vuelve progresivamente menos satisfactorios.
    - Continuing Growth: la funcionalidad de un sistema debe ser incrementada continuamente para mantener la satisfacción del sistema.
    - Increasing Complexity: a medida que un sistema evoluciona, su complejidad se incrementa a menos que se trabaje para evitarlo.
    - Declining Quality: la calidad de un sistema va a ir declinando a menos que se haga un mantenimiento riguroso.
- Prepararse para el cambio:
    - ¿Por qué se pierden oportunidades de negocio?: el ritmo del cambio en los negocios está creciendo exponencialmente. Cambio exponencial implica tiempo de reacción exponencialmente menor.
- Costo del mantenimiento:
    - Mantenimiento: cambios que se producen una vez que el sistema ya fue entregado y está siendo usado por el cliente. Es correctivo, evolutivo, adaptativo, perfectivo, preventivo.
    - Costo: entender código existente: 50% del tiempo de mantenimiento.
    - La incapacidad de cambiar el software de manera rápida y segura implica que se pierden oportunidades de negocio.
- Patrones:

    - Throwaway Code: (código para tirar).
        - Cuando estamos construyendo un sistema solemos empezar por un prototipo.
        - Codificamos rápido para probar una idea, un concepto, con la intención de que después se haga bien.
        - Se hace lo más simple, expeditivo y descartable posible pero el código sigue instalado.
    - Piecemeal Growth: (crecimiento poco a poco).
        - Por más que hayamos comenzado con un diseño de arquitectura elegante, ocurren:
            - Aparición de nuevos requerimientos.
            - Cambios en el entorno/tecnología.
            - Bug fixing.
            - Cambios, cambios y más cambios.
        - Y se agrega código como un "piecemeal growth" continuo que correo las mejoras arquitecturas.
- Diseñar es difícil
    - Los elementos distintivos de la arquitectura de un sistema no surgen hasta después de tener código que funciona. No se trata solo de agregar, sino de adaptar, transformar, mejorar.
    - Construir el sistema perfecto es imposible.
    - Los errores y el cambio son inevitables.
    - Hay que aprender del feedback.
- La iteración es fundamental: los cambios de una iteración a la siguiente pueden involucrar únicamente cambios estructurales entre componentes existentes que no camban la funcionalidad.

**Refactoring**

- Es una transformación que preserva el comportamiento, pero mejora el diseño.
- Refactoring:
    - Es el proceso a través del cual se cambia un sistema de software para **mejorar** la organización, legibilidad, adaptabilidad y mantenibilidad del código luego que ha sido escrito. Y que **no altera** el comportamiento externo del sistema y **mejora** su estructura interna.
    - Características:
        - Implica eliminar duplicaciones, simplificar lógicas complejas, clarificar códigos.
        - ¿Cuándo realizo esto?
            - Una vez que tengo código que funciona y pasa los test.
            - A medida que voy desarrollando: cuando encuentro código difícil de entender (ugly code). Cuando tengo que hacer un cambio y necesito reorganizar primero.
            - Antes de llegar a la bola de barro.
        - Testear después de cada cambio.
- ¿Cómo manejar el cambio?
    - Un mal diseño no es grave hasta que hay que hacer cambios.
    - No podemos prevenir los cambios.
    - El problema no es el cambio sino nuestra incapacidad de manejarlo.
    - Manejar el cambio es difícil:
        - Participan desarrolladores, quienes se preocupan o son afectados.
        - No es fácil descubrir donde cambiar.
        - Es probable que se introduzcan errores.
- ¿Cómo ayuda el Refactoring?
    - Introduce mecanismos que solucionan problemas de diseños.
    - A través de cambios pequeños:
        - Hacer muchos cambios pequeños es más fácil y más seguro que un gran cambio.
        - Cada pequeño cambio pone en evidencia otros cambios necesarios.
- Importancia del Refactoring:
    - Es nuestra única defensa contra el deterioro de software.
    - Facilita la incorporación de código.

    - Permite preocuparse por la generalidad mañana.
    - Permite ser ágil en el desarrollo.
- Refactoring by Fowler
    - Refactoring (sustantivo): cada uno de los cambios catalogados → con un nombre específico y una receta (mecánica).
    - Refactor (verbo): el proceso de aplicar Refactoring.
- Refactorings:
    - Extract Method
        - Se puede usar cuando notamos un código muy largo y notamos que una parte puede estar en otro método.
        - Motivación: métodos largos, métodos muy comentados, incrementar reúso, incrementar legibilidad.
        - En Pharo: botón derecho → Source Code → extract method. Luego poner el nombre e indicar orden de los parámetros.
        - En Pharo (manual): mecánica de varios pasos.
        - A tener en cuenta: testear siempre después de hacer un cambio (si se cometió un error es más fácil corregirlo). Definir buenos nombres.
    - Move Method (para responsabilidad mal asignada)
        - Mover un método de una clase a otra.
        - Motivación: un método está usando o usara muchos servicios que están definidos en una clase diferente a la suya (feature envy).
        - Solución: mover el método a la clase donde están los servicios que usa. Convertir el método original en una simple delegación o eliminarlo.
    - Rename Variable
        - Cambia el nombre de una variable.
        - Todo buen código debería comunicar con claridad lo que hace.
        - Nombres de variables adecuados aumentan la claridad.
        - Solo los buenos programadores escriben código legible por otras personas.
    - Replace condtional with polymorphism
        - Reemplazar un condicional por polimorfismo.
        - Creo una jerarquía, padre abstracto, hijos usa método de acuerdo a la clase.
    - Pull Up Method
        - Si tenemos método repetido en cada subclase:
            - Si los métodos en subclases son iguales → subir directamente.
            - Si los métodos en subclases no son iguales → parametrizar primero.
    - Replace Temp with Query
        - Motivación: usar este Refactoring para:
            - Evitar métodos largos. Las temporales, al ser locales, fomentan métodos largos.
            - Poder usar una expresión desde otros métodos.
            - Antes de un Extract Method, para evitar parámetros innecesarios.
        - Solución:
            - Extraer la expresión en un método.
            - Reemplazar todas las referencias a la variable temporal por la expresión.
            - El nuevo método luego puede ser usado en otros métodos.
- Sobre la performance: la mejor manera de optimizar un programa, primero es escribir un programa bien factorizado y luego optimizarlo, previo profiling.

**Catálogo de Refactoring y Bad Mells**

- ¿Por qué Refactoring es importante?
    - Ganar en la comprensión del código.
    - Reducir el costo del mantenimiento debido a los cambios inevitables que sufrirá el sistema.
    - Facilitar la detección de bugs.

    - La clave: poder agregar funcionalidad más rápido después de refactorizar.
- Permite recrear CLEAN code
    - CLEAN: alta cohesion y bajo acomplamiento.
        - Cohesive,
        - Looselly coupled,
        - Emcapsulated,
        - Assertive,
        - Non-redundant.
    - Pero además: legible.
- Entonces Refactoring:
    - Se toma un código que "huele mal" producto de mal diseño (código diplicado, ilegible, complicado) y se lo trabaja para obtener un buen diseño.
    - ¿Cómo? → moviendo atributos/métodos de una clase a otra. Extrayendo código e un método en otro método. Moviendo código en la jerarquía. Etc.
- Bad Mells
    - Indicios de problemas que requieren la aplicación de refactorings.
    - Algunos Bad Mells pueden ser: duplicate code, large class, long method, data class, feature envy, long parameter list, switch statements, etc.
        - Duplicate Code (Código duplicado):
            - El mismo código o código muy similar aparece en muchos lugares.
            - Problemas: hace el código más largo de lo que necesita ser. Es difícil de cambiar, difícil de mantener. Un bug fix en un clone no es fácilmente propagado a los demás clones.
        - Large Class (Clase grande):
            - Una clase intenta hacer demasiado trabajo.
            - Tiene muchas variables de instancia y/o muchos métodos.
            - Problema: indica un problema de diseño (baja cohesión). Algunos métodos pueden pertenecer a otra clase. Generalmente tiene código duplicado.
        - Long Method (Método largo):
            - Un método tiene muchas líneas de código.
            - Problemas: cuanto más largo es un método, más difícil es entenderlo, cambiarlo y reusarlo.
        - Feature Envy (Envidia de atributo):
            - Un método en una clase usa principalmente los datos y métodos de otra clase para realizar su trabajo (se muestra "envidiosa" de las capacidades de otra clase).
            - Problema: indica un problema de diseño. Idealmente se prefiere que los datos y las acciones sobre los datos vivan en la misma clase.
            - "Feature Envy" indica que el método fue ubicado en la clase incorrecta.
        - Data Class (Clase de datos):
            - Una clase que solo tiene variables y setters/getters para esas variables.
            - Actúa únicamente como contenedor de datos.
            - Problemas: en generar sucede que otras clases tienen métodos con "envidia de atributo". Esto indica que esos métodos deberían estar en la "Data Class". Suele indicar que el diseño es procedural.
        - Condicionales:
            - Cuando sentencias condicionales contienen lógicas para diferentes tipos de objeto.
            - Cuando todos los objetos son instancias de la misma clase, eso indica que se necesitan crear subclases.
            - Problema: la misma estructura condicional aparece en muchos lugares.
        - Long Parameter List:
            - Un método con una larga lista de parámetros es más difícil de entender.

- También es difícil obtener todos los parámetros para pasarlos en la llamada entonces el método es más difícil de reusar.
- La excepción es cuando no quiero crear una dependencia entre el objeto llamador y el llamado.

- Malos Olores – Refactoring a usar
  - Código duplicado → Extract Method, Pull up Method, Form Template Method.
  - Métodos largos → Extract Method, Decompose Conditional, Replace Temp with Query.
  - Clases grandes → Extract Class, Extract Subclass.
  - Muchos parámetros → Replace Parameter with Method, Preserve Whole Objetc, Introduce Parameter Objetc.
  - Cambios divergentes → Extract Class.
  - Shotgun surgery → Move Method/Field.
  - Envidia de atributo → Move Method.
  - Data Class → Move Method.
  - Sentencias Switch → Replace Conditional with Polymorphism.
  - Generalidad especulativa → Collapse Hierarchy, Inline Class, Remove Parameter.
  - Cadena de mensajes → Hide Delegate, Extract Method & Move Method.
  - Middle Man → Remove middle man.
  - Inappropriate Intimacy → move method/field.
  - Legado Rechazado → push down method/field.
  - Comentarios → extract method, rename method.
- Organización de grupos del catálogo de Fowler
  - Composición de métodos:
    - Permiten distribuir el código adecuadamente.
    - Métodos largos son problemáticos.
    - Contiene mucha información y lógica compleja.
    - Refactoring: Extract Method, Inline method, Replace temp with query, Split temporary variable, Replace method with method objetct, Substitute algorithm.
  - Mover aspectos entre objetos:
    - Ayudan a mejorar la asignación de responsabilidades.
    - Refactorings: Move method, Move field, Extract class, Inline class, Remove middle man, Hide delegate.
  - Manipulación de la generalización:
    - Ayudan a mejorar las jerarquías de clase.
    - Refactorings: Push up/Down field, Push up/Down method, Extract subclass/superclass, Collapse hierarchy, Replace inheritance with delegation, Replace delegation with inheritance.
  - Organización de datos:
    - Facilitan la organización de atributos.
    - Refactorings: Self encapsulate field, Encapsulate field/collection, Replace data value with objects, Replace array with objetct, Replace magic number with symbolic constant.
  - Simplificación de expresiones condicionales:
    - Ayudan a simplificar los condicionales.
    - Refactorings: Decompose conditional, Consolidate conditional expression, Consolidate duplicate conditional fragments, Replace conditional with polimorfism.
  - Simplificación de invocación de métodos:
    - Sirven para mejorar la interface de una clase.
    - Refactorings: Rename method, Preserve whole object, Introduce parameter object, Parameterize method.
  - Manipulación de la generalización.
  - Big Refactoring.

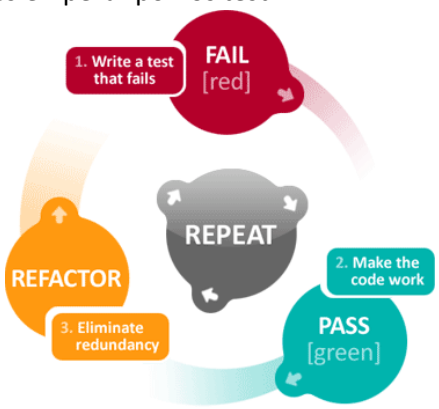
**Refactoring Tools – Automatización del Refactoring**

- Cuando aplicar Refactoring
    - En el contexto de TDD, siempre.
    - Cuando se descubre código con mal olor, aprovechando la oportunidad, dejarlo al menos un poco mejor dependiendo del tiempo que lleve y de lo que esté haciendo.
    - Cuando no puedo entender el código. Aprovechar el momento en que lo logro entender.
    - Cuando encuentro una mejor manera de codificar algo.
- Automatización del Refactoring
    - Refactorizar a mano es demasiado costoso, lleva tiempo y puede introducir errores.
    - Herramientas de Refactoring.
        - Características:
            - Potentes para realizar refactorings útiles.
            - Restrictivas para preservar el comportamiento del programa (uso de precondiciones).
            - Interactivas, de manera que el chequeo de precondiciones no debe ser extenso.
        - Las herramientas solo chequean lo que sea posible desde el árbol de sintaxis y la tabla de símbolos.
        - Las herramientas pueden ser demasiado conservativas (no realizan un Refactoring si no pueden asegurar preservación de comportamiento) o asumir buenas técnicas de programación.
- El Refactoring automático nace con Smalltalk
    - Primera herramienta de Refactoring: Refactoring browser (RB) (en UIUC by Jhon Brant & Don Roberts del grupo de Ralph Johnson).
    - Prácticamente todos los lenguajes tienen herramientas de Refactoring hoy en día, y copian la misma arquitectura/técnica del RB.
    - Más adelante: herramienta Code Citric que detecta code smells.
    - Smalltalk 1ro en: XUnit, Refactoring.

**Test Drive Development (TDD) – Desarrollo guiado por pruebas**

- Método de desarrollo creado por Kent Beck en 2002.
- Concepción del testing pre-TDD:
    - Ultimo paso del desarrollador.
    - Función reservada de un equipo separado de testing o QA (Quality assurance).
    - Equipo de QA escribe toda la documentación sobre la forma de testear cada función.
    - Luego de meses de testing, la lista de errores (bugs) vuelve a los programadores para ser corregidos.
- ¿Qué pasaba en la IS en ese momento?
    - Grupo fuerte de investigación trabajando en MDD: Model Driven Development.
    - Foco en el diseño y documentación.
    - Agile Manifesto firmado en 2001.
    - 1er libro del método de Scrum publicado en 2001.
- Surgen las Metodologías Agiles:
    - También llamadas “Lightweigth” son adaptativas (como opuesto a predictivas) y orientadas a la gente (y no al proceso).
    - Reconocen la gran diferencia entre el diseño y la construcción en la ingeniería “civil”, y el diseño y la construcción de software.
    - Características principales:
        - Mismo grupo de personas para todo el desarrollo que trabajan en un mismo espacio.
        - Comunicación de calidad.
        - Desarrollo iterativo e incremental.
        - Producto funcionando en cada “Build”.
        - Se valora el feedback.
        - Cambios bienvenidos “changes embraced”.
- ¿Por dónde empezar?

    - Si se debe ir tomando de a un requerimiento o pocos por vez para tener un producto funcionando al final de cada iteración, no cuento con un diseño completo para empezar a desarrollar.
    - Entonces empezar por los test.




- TDD combina:
    - Test First Development: escribir el test antes del código que haga pasar el test.
    - Refactoring.
- Objetivo:
    - Pensar en el diseño y que se espera de cada requerimiento antes de escribir código.
    - Escribir código limpio que funcione (como técnica de programación).
- 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.
- ¿Por qué no dejar Testing para el final?
    - Para mantener bajo control un proyecto con restricciones de tiempo ajustadas (permite estimar).
    - Para poder refactorizar rápido y seguro.
    - Para darle confianza al desarrollador de que va por un buen camino.
    - Sirve como una medida de progreso.
- Filosofía de TDD:
    - Vuelco completo al desarrollo de software tradicional. Se escriben primero los tests y luego el código.
    - Se escriben test funcionales para capturar Use Cases que se validad automáticamente.
    - Se escriben test de unidad para enfocarse en pequeñas partes a la vez y aislar los errores.
    - No agregar funcionalidad hasta que no hay un test que no pasa porque esa funcionalidad no existe.
    - Una vez escrito el test, se codifica lo necesario para que el test pase.
    - Pequeños pasos: un test, un poco de código.
    - Una vez que los test pasan, se refactoriza para asegurar que se mantenga una buena calidad en el código.
- Algunas reglas TDD:
    - Diseñar incrementalmente teniendo código que funciona como feedback para ayudar en las decisiones entre iteraciones.
    - Los programadores escriben sus propios test. 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 de TDD:
    - TDD asume la presencia de herramientas de testing (como las de la familia xUnit).
    - Sin herramientas que automaticen el testing, TDD es prácticamente imposible.
    - El ambiente de desarrollo debe proveer respuesta rápida ante cada cambio.
- ¿Qué logramos con TDD?

    - Diseño simple y limpio. Desarrollar más rápido. Saber cuándo terminamos. Confianza para el desarrollo. Coraje para refactorizar. Documentación practica que evoluciona naturalmente. Incrementar la calidad del software en dos aspectos:
        - Que el software este construido correctamente.
        - Que el software construido sea el correcto.
- Problemas y Respuestas
    - Problemas:
        - Unit testing infinito: por cada método público.
        - Test coupling: al estar los tests atados a la implementación.
    - Respuestas:
        - Saber porque se testea algo y a qué nivel debe testearse.
        - El objetivo de testear es encontrar bugs.
        - Testear tanto como sea el riesgo del artefacto.
- Reducir el riesgo del proyecto:
    - Puede hacerse de distintas formas:
        - Reduciendo la cantidad de bugs.
        - Previniendo la aparición de bugs.
        - Acercando el SUT a las necesidades del usuario.
        - Testeando el SUT bajo condiciones extremas.

**Testing de Unidad**

- Tamaño de los métodos de Testing
    - Postura más purista: verificar una sola condición por cada test.
    - Ventaja para detectar errores: cuando un test falla se puede saber con precisión que esta mal con el SUT.
    - Un test que verifica una única condición ejecuta un solo camino en el código del SUT (cobertura).
    - Desventaja: el costo de testear cada camino de ejecución es demasiado alto. Cobertura de 100% es inviable.
- Tamaño de las clases de Testing
    - Una subclase de TestCase por cada clase → TestInteger
    - Una clase testcase por cada método (feature) → TestIntegerFactorial
    - Una clase testcase por cada fixture → TestLargeIntegerFactorial
- ¿Qué se hace en Fixture Setup?
    - La lógica del fixture setup incluye:
        - El código para instanciar el SUT.
        - El código para poner el SUT en el estado apropiado.
        - El código para crear e inicializar todo aquello de lo que el SUT depende o que le va a ser pasado como argumento.
- Aislar el SUT
    - Las distintas funcionalidades del SUT en muchos casos dependen entre sí o de componentes ajenos al SUT.
    - Cuando se producen cambios en los componentes de los que depende el test, es posible que este último empiece a fallar.
    - Al testear funcionalidades del SUT es preferible no depender de componentes del sistema ajenos al tests.
    - ¿Qué debemos hacer? → Test Doubles (también conocidos como “Mock Objects”).
- Mock Objects
    - Son simuladores que imitan el comportamiento de otros objetos de manera controlada.
    - 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 envié los mensajes esperados y devuelva un valor por defecto.
        - Test Spy: test stub + registro de outputs.

        - Mock Object: test stub + verification of outputs.
        - Fake Object: imitación. Se comporta como el modulo real (protocolos, tiempo de respuesta, etc.)
    - ¿Cuándo usar Test Doubles?
        - Cuando el objeto real es un objeto complejo que:
            - Retorna resultados no-determinístico (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.
            - Tiene dependencias con otros objetos y necesita ser asilado para testearlo como unidad.
- Reglas del testing
    - Mantener los tests independientes entre sí.
    - Un buen test es simple, fácil de escribir y mantener (que requiera mínimo mantenimiento a medida que el sistema evoluciona).
    - El objetivo de testear es encontrar bugs.
    - Limitaciones del testing: no encuentra todos los errores y no puede comprobar la ausencia de errores.
- Testing con BBDD
    - Es mejor testear sin usar BBDD:
        - Fake Database: es un tipo de test doublé. Se crea usando Fresh Fixture.
        - Se usa para testear la lógica de negocio por separado.
    - Cuando hace falta testear la BBDD:
        - Database Sandbox: una copia para cada desarrollador evita el Test Run War.
        - Es importante testear la capa de acceso a los datos.
        - JUnit tiene extensiones para BBDD (DbUnit) que por ejemplo permite fácilmente comparar un resultado de un join con una tabla plana en XML.
- Testing de Aplicaciones Web
    - ¿Qué testear? → funcionalidad, usabilidad, interface con el servidor y la BBDD, compatibilidad (browser, SOs), performance, seguridad.
    - Herramientas → Selenium, TestComplete, etc.

**Introducción a Patrones de Diseño**

- La importancia de la experiencia de diseño:
    - Diseñar software es difícil (aun usando buenas técnicas). Necesitamos que sea adaptable, extensible, etc.
    - La experiencia es insustituible.
- Desing Patterns (Patrones de Diseño)
    - Originados en la arquitectura (Alexander).
    - Según Alexander: “un patrón describe un problema que ocurre una y otra ve en un contexto y describe la parte central de la solución a ese problema de tal manera de que puede ser usando millones de veces sin hacer lo mismo dos veces”.
    - Patrones, definiciones:
        - Un patrón es un par problema-solución.
        - Los patrones tratan con problemas recurrentes y buenas soluciones a esos problemas.
        - Los patrones deben incluir una regla, ¿Cuándo aplico el patrón?
    - Patrón de Diseño según GOF:
        - “Un patrón de diseño nombra, abstrae e identifica los aspectos claves de una estructura de diseño común que la hacen útil para crear un diseño orientado a objetos reusable. Un patrón de diseño, identifica las clases participantes, sus instancias, los roles y las colaboraciones, y la distribución de responsabilidades. Cada patrón se enfoca en un problema de diseño orientado a objetos particular. Describe cuando se aplica, cuando puede ser aplicado en vistas de otras restricciones, y las consecuencias que tiene usarlo.”

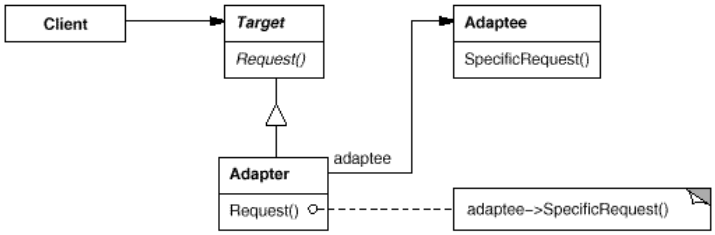
- Clasificación de patrones:
  - De acuerdo a la actividad: patrones de diseño de software, de testing, de proceso, de interacción, de interfaz.
  - De acuerdo al nivel de abstracción: patrones de análisis, de arquitectura, de diseño, de programación.
  - De acuerdo al dominio de aplicación: patrones que aparecen en dominios como el financiero, comercio electrónico, salud, sistema de tiempo real.
  - De acuerdo al propósito: patrones creacionales, estructurales, de comportamiento.

- Resumiendo
  - Un patrón describe un problema recurrente y su solución.
  - Necesitamos alguna formalización como para que esto no sea "simplemente" un conjunto de consejos profesionales.
- Template → descripción de un patrón, según GOF:
  - Nombre (name).
  - Intención (intent): que busca el patrón? Problema a resolver.
  - Motivación (motivation): solución.
  - Aplicabilidad (aplicability): regla, usa esta solución para este problema.
  - Estructura (structure): descripción en termino UML de la solución.
  - Participantes (participants).
  - Colaboración (collaborations).
  - Consecuencias (consequences): que paso cuando uso el patrón? Que gano? Que pierdo?.
  - Implementación (implementation).
  - Código (code).
  - Casos conocidos (Kwonw Uses).
  - Patrones relacionados (related patterns)

**Patrones de Diseño**

**Pattern Adapter** (Estructural)

- Intención: "construir" 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.
- Aplicabilidad: use el adapter cuando usted quiere usar una clase existente y su interfaz no es compatible con lo que precisa.
- Estructura:

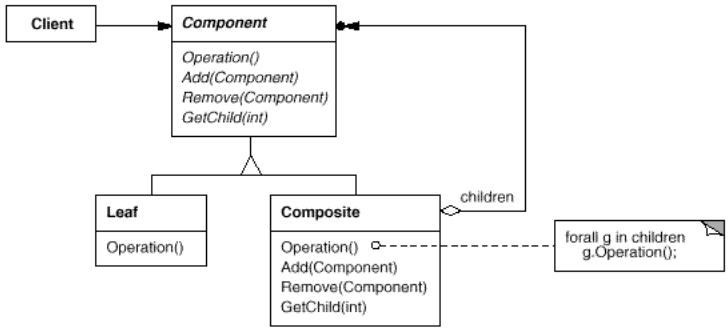



- Participantes:
  - Target: define la interfaz del dominio que usa el cliente.
  - Client: colabora con objetos conformes a la interfaz de destino.
  - Adaptee: define una interfaz existente que necesita adaptando.
  - Adapter: adapta la interfaz de Adaptee a la interfaz de destino.
- Colaboración: el cliente colabora con objetos Adapter quienes a su vez lo hacen con instancias de Adaptee.

**Pattern Composite** (Estructural)

- 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.
- Aplicabilidad: use el patrón composite cuando quiere representar jerarquías parte-todo de objetos. Y cuando quiere que los objetos "clientes" puedan ignorar las diferencias entre composiciones y objetos individuales. Los clientes trataran a los objetos atómicos y compuestos uniformemente.

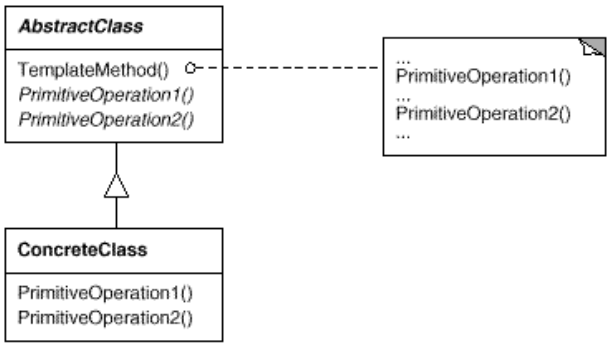
- Estructura:




- Participantes:
    - Component: declara la interfaz para los objetos de la composición. Implementa comportamientos default para la interfaz con todas las clases. Declara la interfaz para definir y acceder "hijos". Como opcional define una interfaz para acceder al "padre" de un componente en la estructura recursiva y la implementa si es apropiado.
    - Leaf: representa arboles "hojas" en la composición, las hojas no tienen "hijos". Define el comportamiento de objetos primitivos en la composición.
    - Composite: define el comportamiento para componentes con "hijos". Contiene las referencias a los "hijos". Implementa operaciones para manejar "hijos".
- Consecuencias:
    - El patrón composite define jerarquías de clases consistentes de objetos primitivos y compuestos. Los objetos primitivos puede componerse en objetos complejos, los que a su vez pueden componerse y así recursivamente. En cualquier lugar donde un cliente espera un objeto simple, puede aparecer un compuesto.
    - Simplifica los objetos cliente. Los clientes pueden tratar estructuras compuestas y objetos individuales uniformemente. Los clientes usualmente no saben (y no deberían preocuparse) acerca de si están manejando un compuesto o un simple. Esto simplifica el código del cliente, porque evitar tener que escribir código taggeado con estructura de decisión sobre las clases que definen la composición.
    - Pero:
        - Puede hacer difícil restringir las estructuras de composición cuando hay algún tipo de conflicto (por ejemplo ciertos compuestos pueden armarse solo con cierto tipo de atómicos).

**Pattern Template Method** (Comportamiento)

- 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.
- Estructura:

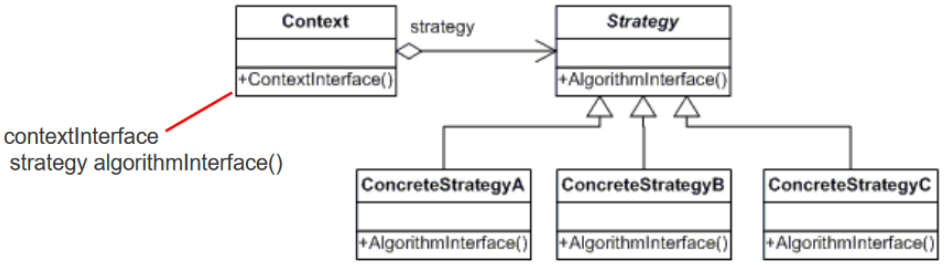



- Las operaciones primitivas también se denominan método hook.

- Observen que yo puedo implementar una operación primitiva en una subclase la cual será invocada por un código pre-existente: el template method.

**Pattern Strategy** (Comportamiento)

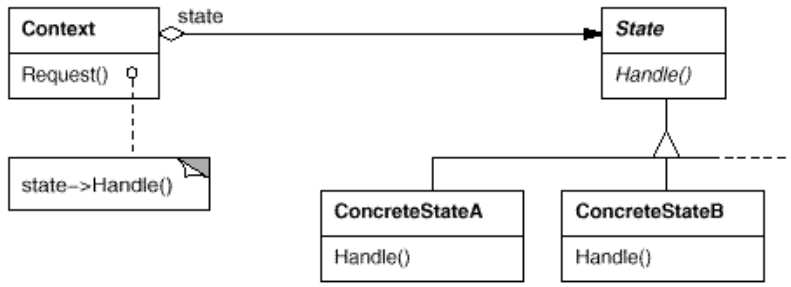
- Intención: desacoplar un algoritmo del objeto que lo utiliza. Permitir cambiar el algoritmo que un objeto utiliza en forma dinámica. Brindar flexibilidad para agregar nuevos algoritmos que lleven a cabo una función determinada.
- Aplicabilidad: existen muchos algoritmos para llevar a cabo una tarea. No es deseable codificarlos todos en una clase y seleccionar cual utilizar por medio de sentencias condicionales. Es necesario cambiar el algoritmo en forma dinámica.
- Estructura:




- Consecuencias:
  - Alternativa a subclasificar el contexto, para permitir que se pueda cambiar dinámicamente.
  - Desacopla al contexto de los detalles de implementación de las estrategias.
  - Se eliminan los condicionales.
  - Overhead en la comunicación entre contexto y estrategias.
- Implementación: el contexto debe tener en su protocolo métodos que permitan cambiar la estrategia. Parámetros entre el contexto y la estrategia.

**Pattern State** (Comportamiento)

- Intención: modificar el comportamiento de un objeto cuando su estado interno se modifica. Externamente parecería que la clase del objeto ha cambiado.
- Aplicabilidad: usamos el patrón State cuando 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).
- Detalles: desacoplar el estado interno del objeto en una jerarquía de clases. Cada clase de la jerarquía representa un estado concreto en el que puede estar el objeto. Todos los mensajes del objeto que dependan de su estado interno son delegados a las clases concretas de la jerarquía (polimorfismo).
- Estructura:

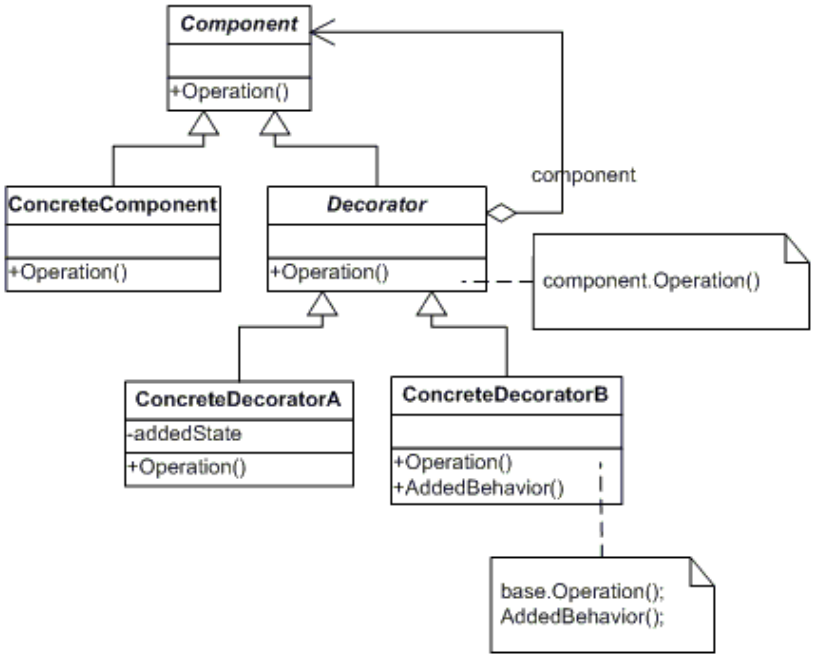



- Participantes:
  - Context: define la interfaz que conocen los límites. Mantiene una instancia de alguna clase de ConcreteState que define el estado corriente.
  - State: define la interfaz para encapsular el comportamiento de los estados de Context.
  - ConcreteState subclases: cada subclase implementa el comportamiento respecto al estado específico.

- Consecuencias: localiza el comportamiento relacionado con cada estado. Las transiciones entre estados son explicitas. En el caso que los estados no tengan variables de instancia pueden ser compartidos.

**Pattern Decorator** (Estructural)

- Objetivo: agregar comportamiento a un objeto dinámicamente y en forma transparente.
- Problema: cuando queres agregar comportamiento extra a algunos objetos de una clase puede usarse herencia. El problema es cuando necesitamos que el comportamiento se agregue o quite dinámicamente, porque en ese caso los objetos deberían "mutar de clase". El problema que tiene la herencia es que se decide estáticamente.
- Aplicabilidad: usar Decorator para:
    - Agregar responsabilidades a objetos individualmente y en forma transparente (sin afectar otros objetos).
    - Quitar responsabilidades dinámicamente.
    - Cuando subclasificar es impráctico.
- Solución: definir un Decorator (o "wrapper") que agregue el comportamiento cuando sea necesario.

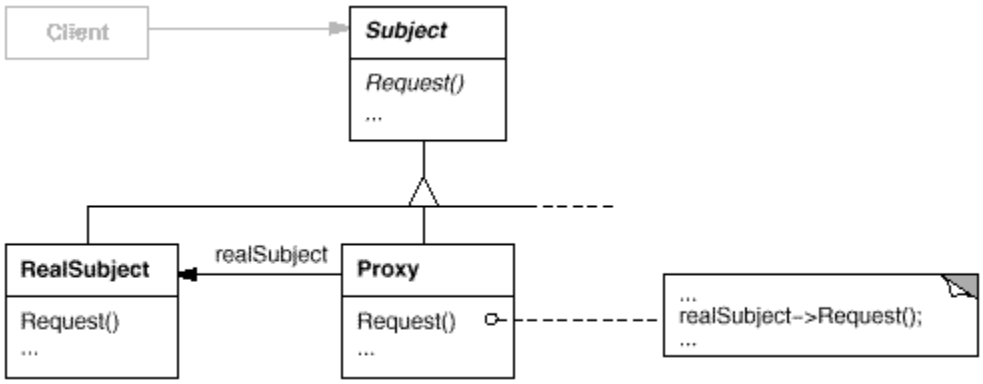



- Consecuencias:
    - + → Permite mayor flexibilidad que la herencia.
    - + → Permite agregar funcionalidad incrementalmente.
    - - → Mayor cantidad de objetos, complejo para depurar.
- Implementación:
    - Misma interface entre componente y decorador.
    - No hay necesidad de la clase Decorator abstracta.
    - Cambiar el "skin" vs cambiar sus "guts".
- Decorator vs Adapter (Wrappers)
    - Ambos patrones "decoran" el objeto para cambiarlo.
    - Decorator *preserva* la interface del objeto para el cliente.
    - Adapter *convierte* la interface del objeto para el cliente.
    - Decorators pueden y suelen anidarse.
    - Adapters no se anidan.
- Decorator vs Strategy
    - ¿en que se parecen?
        - Propósito: permitir que un objeto cambie su funcionalidad dinámicamente (agregando o cambiando el algoritmo que utiliza).
    - ¿en qué se diferencian?

- Estructura: el Strategy cambia el algoritmo por dentro del objeto, el Decorator lo hace por fuera.

**Pattern Proxy** (Estructural)

- Propósito: proporcionar un intermediario de un objeto para controlar su acceso.
- Aplicabilidad: cuando se necesita una referencia a un objeto más flexible o sofisticada.
- Solución:
  - Colocar un objeto intermedio que respete el protocolo del objeto que esta reemplazando.
  - Algunos mensajes se delegaran en el objeto original. En otros casos puede que el Proxy colabore con el objeto original o que reemplace su comportamiento.




- Aplicaciones del Proxy:
  - Virtual proxy: Demorar la construcción de un objeto hasta que sea realmente necesario.
  - Protection proxy: Restringir el acceso a un objeto por seguridad.
  - Remote proxy: Implementación de objetos distribuidos.
    - Para acceder a objetos que se encuentran en otro espacio de memoria, en una arquitectura distribuida.
    - El proxy empaqueta el Request, lo envía a través de la red al objeto real, espera la respuesta, desempaqueta la respuesta y retorna el resultado.
    - En este contexto el proxy suele utilizarse con otro objeto que se encarga de encontrar la ubicación del objeto real. Este objeto se denomina **Broker**, del patrón de su mismo nombre.
- Implementación:
  - Redefinir todos los mensajes del objeto real ¿?
  - Proxy no siempre necesita conocer la clase del objeto real.
- Implementación de Proxy usando reflexión:
  - Method look-up.
  - #doesNotUnderstand: aMessage
  - Cuando a un objeto se le envía un mensaje que no implementa, la máquina virtual le envía el mensaje #doesNotUnderstand: al objeto con una "reificación" del mensaje como argumento.
  - El mensaje (instancia de Message) contiene al selector y un Array de los argumentos.
  - En el método #doesNotUnderstand: podríamos examinar el contexto en el que ocurrió el error, cambiarlo y continuar la ejecución.
  - → en vez de reimplementar en Proxy todos los mensajes, solo se define #doesNotUnderstand:
  - → cada envío de mensaje a instancias de Proxy termina en #doesNotUnderstand:, donde el objeto puede manipular el mensaje para por ejemplo, enviárselo al objeto real.
- Entendiendo que es reflexión
  - Un programa reflexivo es aquel que puede razonar sobre sí mismo, es decir que puede ***observarse*** y ***cambiarse*** dinámicamente.
  - Es aquel que puede observar su propia ejecución (***introspección***) e incluso cambiar la manera en que se ejecuta (***intercesión***).
  - Requiere poder expresar y manipular el estado de la ejecución como datos: ***reification.***