Patrones de diseño

Patrón: par de problema-solución, donde el problema es recurrente y las soluciones son buenas. Incluyen una regla que indica cuando aplicar el patrón

Nombra, abstrae e identifica los aspectos claves de una estructura de diseño común, que la hacen útil para crear un diseño OO reusable. Identifica las clases participantes, sus instancias, los roles, las colaboraciones y la distribución de responsabilidades. Cada patrón se enfoca en un problema de diseño OO particular, describe cuando se aplica, restricciones y consecuencias de usarlo.

En el libro de gamma, los patrones se dividen en 3 categorías:

Nombre	**Descripción**	**Algunos patrones**
Creacionales	Organizan la instanciación de objetos	
Estructurales	Describen formas de estructurar objetos complejos y sus relaciones	Adapter, composite, decorator, proxy
De comportamiento	Organizan la asignación de responsabilidades	Template method, strategy, state

(Descripción de cada patrón en el cuaderno)

Refactoring a patrones

La relación entre refactoring y patrones es tenida en cuenta incluso en el libro de patrones gamma. Fowler dijo que “los patrones son los lugares donde queremos estar y los refactorings los caminos para llegar ahí desde otro lado”

En el libro ‘refactoring a patrones’ se dice que los patrones, con todas las ventajas que conocemos, a veces pueden llevarnos a un extremo indeseado. La sobre ingeniería, osea construir software mucho más sofisticado del que se necesita, es tan peligrosa como la poca ingeniería, que significa construir software con un diseño pobre.

La poca ingeniería ya vimos que tiene causa el los tiempos ajustados con los que se cuenta para desarrollar, teniendo como consecuencia codigo poco flexible, reusable, extensible, etc Por otro lado, las causas de una sobre ingeniería pueden ser:

- Para acomodar futuros cambios, pero hay que tener en cuenta que no se puede predecir el futuro
- Para no quedar inmerso y acarrear un mal diseño. Sin embargo, a la larga, el encanto de los patrones puede hacer que perdamos de vista formas más simples de

escribir código.

Las consecuencias son que el código sofisticado, complejo, se instala y complica el mantenimiento. Como nadie lo entiende, nadie lo quiere tocar y se generan copias, código duplicado.

El poder del **refactoring** es que permite aplicar un diseño simple que se ajuste a los requerimientos que se tienen actualmente y, si en un futuro el problema crece o se vuelve complejo, aplicar un **patrón**.

En el libro de patrones de gamma, se explica cómo crear nuevos diseños, no cómo aplicar patrones para solucionar problemas en código ya existente.
En su libro refactoring to patterns, Joshua Kerievsky establece algunos *refactorings que te llevan a tener como resultado patrones* que ya vimos en el libro de gamma. Describe de qué manera llegar a ese punto.

- Form Template Method
- Extract Adapter
- Replace Implicit Tree with Composite
- Replace Conditional Logic with Strategy
- Replace State-Altering Conditionals with State
- Move Embellishment to Decorator

En el libro de refactoring a patrones, se describen refactorings más complejos que los que ya vimos, pero lo bueno es que se explican como una serie de pasos que el autor encontró para llevar un diseño a un patrón y, esos pasos, son por lo general refactorings más pequeños/sencillos, los que ya vimos antes.

Form template method

Descripción: dos o más métodos en subclases realizan pasos similares, en el mismo orden, pero los pasos son distintos.

Solución: Generalizar los métodos extrayendo sus pasos en métodos con el mismo nombre, y luego subir a la superclase común el método generalizado para formar un Template Method.

Mecánica:

1. Encontrar el método que es similar en todas las subclases y extraer sus partes en: métodos idénticos (mismo nombre y cuerpo en las subclases) o métodos únicos (distinto nombre y cuerpo)
2. Aplicar “PullUp Method” para los métodos idénticos.
3. Aplicar “RenameMethod” sobre los métodos únicos hasta que el método similar quede con cuerpo idéntico en las subclases.
4. Compilar y testear después de cada “rename”.
5. Aplicar “RenameMethod” sobre los métodos similares de las subclases (esqueleto).
6. Aplicar “PullUp Method” sobre los métodos similares.

7. Definir métodos abstractos en la superclase por cada método único de las subclases.
8. Compilar y testear

Ventajas:
- Elimina código duplicado en las subclases moviendo el comportamiento invariante a la superclase.
- Simplifica y comunica efectivamente los pasos de un algoritmo genérico
- Permite que las subclases adapten fácilmente un algoritmo

Desventajas:
- Complica el diseño cuando las subclases deben implementar muchos métodos para sustanciar el algoritmo. Cuando en el template hay algunos pasos que para algunas subclases no tienen sentido y tienen que definirlos igual, para que no hagan nada, con el fin de que el template funcione. Se agregan métodos a las subclases que si no fuera por el template no estarían. Cuando se da mucho esta situación, entonces el template deja de funcionar bien porque quizás la jerarquía esté mal definida.

<u>Replace conditional logic with strategy</u>

Descripción: Existe lógica condicional en un método que controla qué variante ejecutar entre distintas posibles

Solución: Crear un Strategy para cada variante y hacer que el método original delegue el cálculo a la instancia de Strategy

Mecánica:
1. Crear una clase Strategy.
2. Aplicar “Move Method” para mover el cálculo con los condicionales del contexto al strategy.
    a. Definir una var. de instancia en el contexto para conocer al strategy y un método para instanciarlo
    b. Dejar un método en el contexto que delegue
    c. Elegir los parámetros necesarios para pasar al strategy
    d. Compilar y testear.
3. Aplicar “Extract Parameter” en el código del contexto que inicializa un strategy concreto, para permitir a los clientes setear el strategy. Compilar y testear.
4. Aplicar “Replace Conditional with Polymorphism” en el método del Strategy.
5. Compilar y testear con distintas combinaciones de estrategias y contextos.

Nota: Una alternativa a lo anterior es, si no hay muchas combinaciones de Strategies y contextos, aislar el código del cliente de preocuparse de cómo instanciar las subclases de Strategy. Se usa el refactoring Encapsulate Classes with Factory: definir un método en el contexto que retorna una instancia del mismo con el strategy correspondiente, por cada subclase de Strategy

Ventajas:

- Clarifica los algoritmos al reducir o remover la lógica condicional.
- Simplifica una clase moviendo variaciones de un algoritmo a una jerarquía separada
- Permite reemplazar un algoritmo por otro en runtime

Desventajas:

- Complica el diseño cuando se podría solucionar con subclases o simplificando los condicionales

Replace State-Altering Conditionals with State

Descripción: Las expresiones condicionales que controlan las transiciones de estado de un objeto son complejas

Solución: Reemplazar los condicionales con States que manejen estados específicos y las transiciones entre ellos

Motivación:

- Obtener una mejor visualización con una mirada global, de las transiciones entre estados.
- Cuando la lógica condicional entre estados dejó de ser fácil de seguir o extender
- Cuando aplicar refactorings más simples, como “Extract Method” o “Consolidate Conditional Expressions” no alcanzan

Mecánica:

Hay 2 formas de llevar a cabo el primer paso, dependiendo de como se representa el estado actual en el contexto:

1. Si hay una sola var. de instancia que se compara con distintas constantes then “Replace Type-Code with Class”
    a. Aplicar “Self-Encapsulate Field”.
    b. Crear una nueva clase: superclase del State.
    c. Agregar una var. de instancia en la clase contexto para el estado y su setter.
    d. Cambiar los setters. C&T
    e. Cambiar los getters. C&T
    f. Borrar la vieja var de instancia

1. Si hay más de una var. de instancia que mantiene el estado, por ejemplo booleandos con cada estado posible, then “Extract Class”
    a. Crear una nueva clase: superclase del State.
    b. Agregar una v.i. en la clase contexto para el estado y su setter.
    c. Aplicar “Move Field”[F]. C&T
    d. Aplicar “Move Method”[F]. C&T

2. Aplicar “Extract Subclass” [F] para crear una subclase del State por cada uno de los estados en los que la clase contexto puede entrar.

3. Por cada método de la clase contexto con condicionales que cambiar el valor del estado, aplicar “Move Method” hacia la superclase de State.
4. Por cada estado concreto, aplicar “Push down method” para mover de la superclase a esa subclase los métodos que producen una transición desde ese estado. Sacar la lógica de comprobación que ya no hace falta.
5. Dejar estos métodos como abstractos en la superclase o como métodos por defecto.

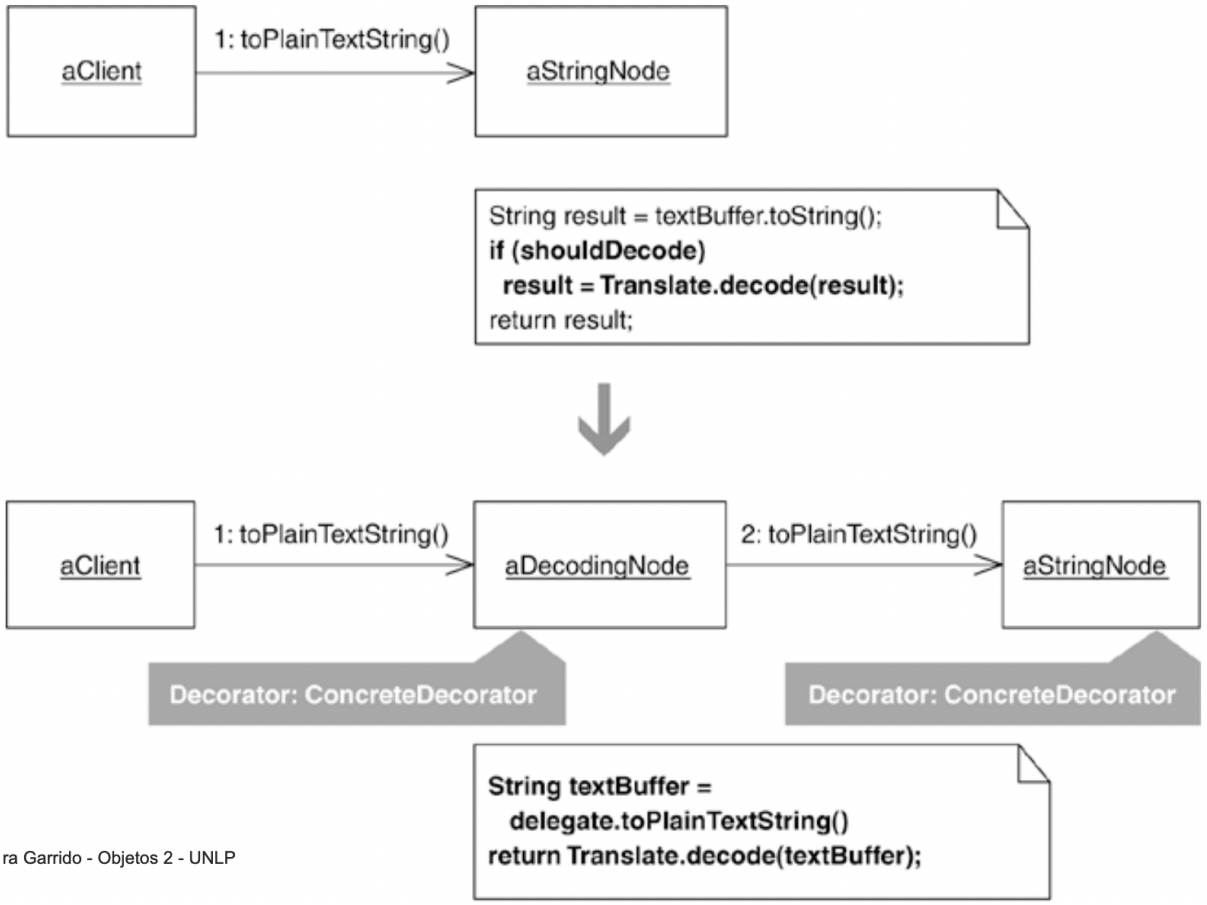
Ventajas:

- Reduce o remueve la lógica condicional de cambio de estado.
- Simplifica la lógica compleja de transiciones.
- Provee una mejor visualización de alto nivel de los posibles estados y transiciones.

Desventajas:

- Complica el diseño cuando la lógica de transición de estados ya es fácil de seguir

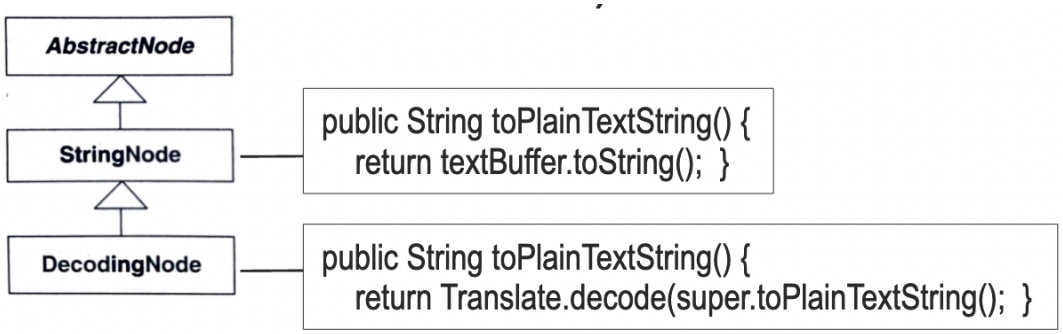
Move Embellishment to Decorator



En este ejemplo, se tiene un booleano shouldDecode adentro del objeto aStringNode, que verifica si, después de pasarlo a string, hay que decodificarlo. Una solución aplicada es sacar esa variable del objeto y que se maneje con el decorator aDecodingNode, que delega el mensaje a su componente (aStringNode) y cuando este termina, lo decodifica.

Mecánica:

1. Identificar la superclase (or interface) del objeto a decorar (clase Component del patrón). Si no existe, crearla.
2. Aplicar Replace Conditional Logic with Polymorphism (crea decorator como subclase del decorado)



En este punto quizás alcance para algún problema y es posible terminar acá. Sino:

3. Aplicar Replace Inheritance with delegation (decorator delega en decorado como clase “hermana”)
4. Aplicar Extract Parameter en decorator para asignar decorado

Otras cosas importantes del libro de refactoring a patrones:

- Importancia de TDD y refactoring continuo: “lean, iterative and disciplined style of programming that maximizes focus, relaxation and productivity”
- Many eyes: importancia del code review
- Importancia del refactoring en la evolución de la arquitectura para la generación de frameworks
- Refactorings compuestos y test-driven