

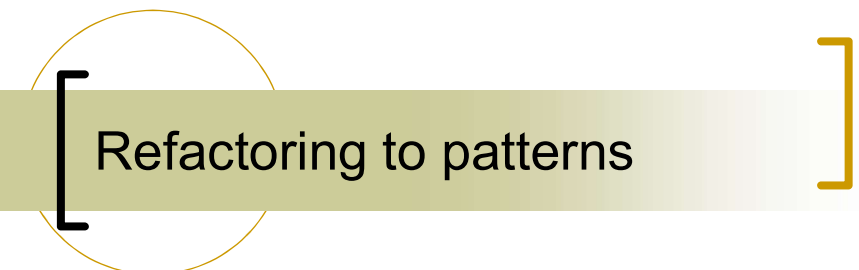
Refactoring de código

Mejorar el diseño de código existente

Alejandra Garrido – Objetos 2

Algunos refactorings del catálogo de Fowler

- *Composición de métodos*
 - Extract Method
 - Replace Temp with Query
- *Mover aspectos entre objetos*
 - Move Method
- *Simplificación de expresiones condicionales*
 - Replace Conditional with Polymorphism
- *Manipulación de la generalización*
 - Pull Up Method
- *Organización de datos*
 - Encapsulate Field / Self Encapsulate Field
- *Simplificación en la invocación de métodos*
 - Rename Method
 - Preserve Whole Object



Refactoring to patterns

Mejoras de diseño complejas



Relación entre patrones y refactorings

- “Design patterns provide targets for your refactorings.”
 - Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- “Patterns are where you want to be; refactorings are ways to get there from somewhere else.”
 - Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley, 2000.

[Refactoring to Patterns]

Joshua Kerievsky.
Refactoring to Patterns.
Addison Wesley, 2005.



La sobre-ingeniería es tan peligrosa como la poca ingeniería.

- Sobre-ingeniería (over-engineering) significa construir software más sofisticado de lo que realmente necesita ser.
 - vs
- Poca ingeniería (under-engineering) significa producir software con un diseño pobre.
 - ¿Por qué se hace? ¿Consecuencias?

5

Alejandra Garrido - Objetos 2 - UNLP

[Over-Engineering]

- Por qué se hace?
 - Para acomodar futuros cambios (pero no se puede predecir el futuro)
 - Para no quedar inmerso y acarrear un mal diseño.
- Consecuencias:
 - El código sofisticado se queda y complica el mantenimiento.
 - Nadie lo entiende. Nadie lo quiere tocar.
 - Como otros no lo entienden generan copias, código duplicado.

Alejandra Garrido - Objetos 2 - UNLP

6

[La panacea de los patrones]

- 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.

7

Alejandra Garrido - Objetos 2 - UNLP

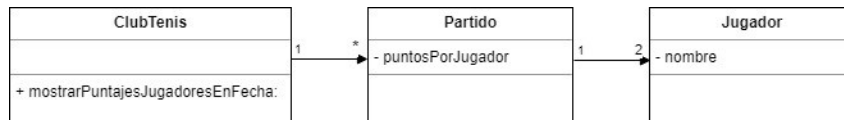
[Refactoring to Patterns]

- Form Template Method
- Extract Adapter
- Replace Implicit Tree with Composite
- Extract Composite
- Replace Conditional Logic with Strategy
- Inline Singleton

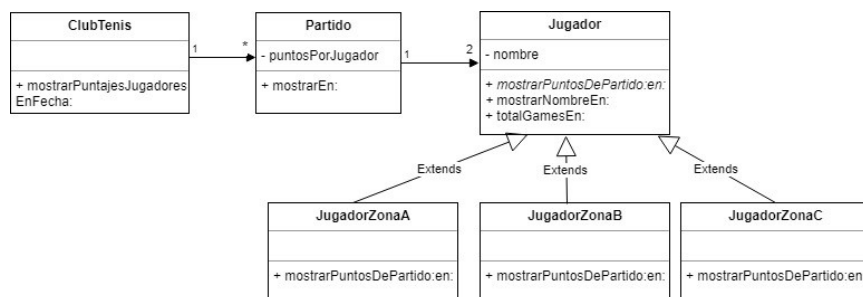
8

Alejandra Garrido - Objetos 2 - UNLP

[Volviendo al ejemplo de la clase de refactoring: empezamos así...]



[Terminamos así. Cómo?]



[En JugadorZonaA...]

```
>>mostrarPuntosDePartido: unPartido en: unStream
| totalGames |
self mostrarNombreEn: unStream.
totalGames := 0.
(unPartido puntosDelJugador: self)
do: [ :gamesDelSet |
    unStream nextPutAll: gamesDelSet asString, ';'.
    totalGames := totalGames + gamesDelSet ].
unStream nextPutAll: ' Puntos del partido: '.
unStream nextPutAll: (totalGames * 2) asString.

>>mostrarNombreEn: unStream
unStream
    nextPutAll: 'Puntaje del jugador: ';
    nextPutAll: self nombre;
    nextPutAll: ' '.
```

Alejandra Garrido - Objetos 2 - UNLP

11

[Después de Pull Up Method y Replace Temp with Query]

```
JugadorZonaA>>mostrarPuntosDePartido: unPartido en: unStream
self mostrarNombreEn: unStream.
(unPartido puntosDelJugador: self)
do: [ :gamesDelSet |
    unStream nextPutAll: gamesDelSet asString, ';'].
unStream nextPutAll: ' Puntos del partido: '.
unStream nextPutAll: (self totalGamesEn: unPartido * 2) asString.
```

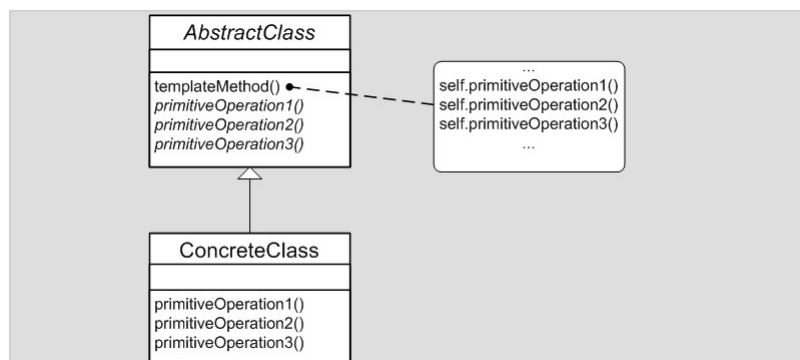
Alejandra Garrido - Objetos 2 - UNLP

12

[Ahora]

- Sigue habiendo código duplicado en las subclases.
- Qué patrón puedo usar para solucionarlo?

[Patrón Template Method]



Propósito de un patrón vs. Problema que soluciona

- Propósito del patrón Template Method:
 - *Definir el esqueleto de un algoritmo en una operación, y diferir algunos pasos a las subclases. Template Method permite que las subclases redefinan algunos pasos de un algoritmo sin cambiar la estructura del algoritmo.*
- aunque el problema que soluciona es que:
 - *reduce o elimina el código repetido en métodos similares de las subclases en una jerarquía.*

15

Alejandra Garrido - Objetos 2 - UNLP

Refactoring “Form Template Method”

- Dos o más métodos en subclases realizan pasos similares en el mismo orden, pero los pasos son distintos.



- Generalizar los métodos extrayendo sus pasos en métodos de la misma signatura, y luego subir a la superclase común el método generalizado para formar un Template Method.

16

Alejandra Garrido - Objetos 2 - UNLP

Refactoring “Form Template Method”. 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) Aplicar “**Pull Up Method**” para los métodos idénticos.
- 3) Aplicar “**Rename Method**” 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 “**Rename Method**” sobre los métodos similares de las subclases (esqueleto).
- 6) Aplicar “**Pull Up Method**” sobre los métodos similares.
- 7) Definir métodos abstractos en la superclase por cada método único de las subclases.
- 8) Compilar y testear

17

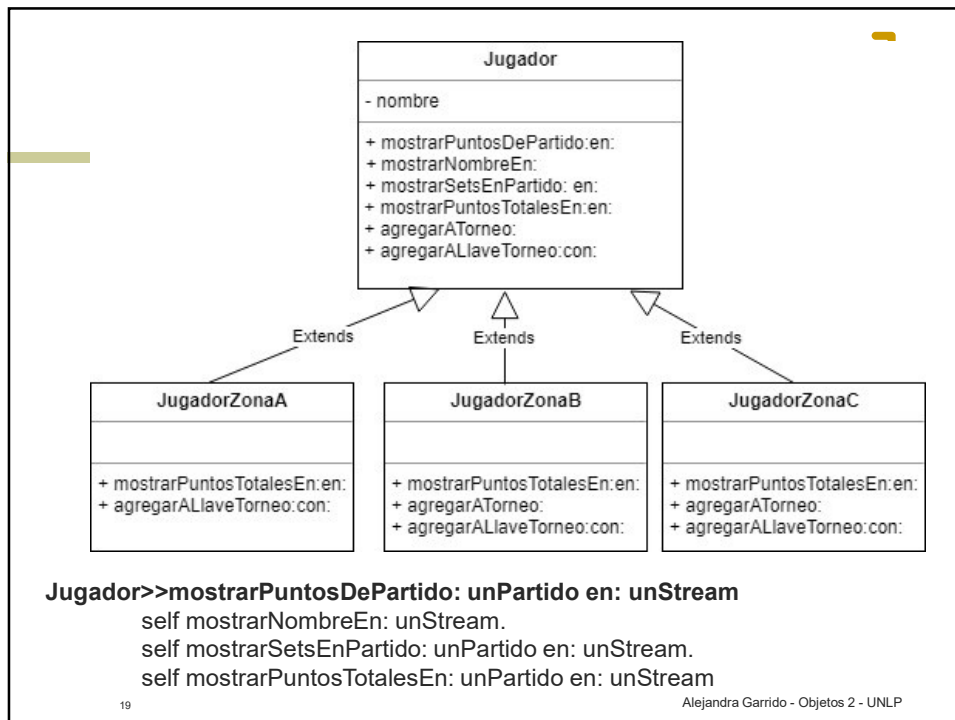
Alejandra Garrido - Objetos 2 - UNLP

Form Template Method: Pros y Contras

- 👍 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
- 👎 Complica el diseño cuando las subclases deben implementar muchos métodos para sustanciar el algoritmo

18

Alejandra Garrido - Objetos 2 - UNLP



[
]

Otro requerimiento

- El jugador puede cambiar de zona

Alejandra Garrido - Objetos 2 - UNLP 20

Replace Conditional Logic with 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

21

Alejandra Garrido - Objetos 2 - UNLP

Replace Conditional Logic with Strategy. Mecánica

- 1) Crear una clase Strategy.
- 2) Aplicar "*Move Method*" para mover el cálculo con los condicionales del contexto al strategy.
 - 1) Definir una v.i. en el contexto para conocer al strategy y un método para instanciarlo
 - 2) Dejar un método en el contexto que delegue
 - 3) Elegir los parámetros necesarios para pasar al strategy (el contexto entero? Sólo algunas variables?)
 - 4) 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.

22

Alejandra Garrido - Objetos 2 - UNLP

[Extract Parameter]

```
Jugador>>newConNombre: unString fechaDeNacimiento: aDate  
    ^super new initializeConNombre: unString fechaNac: aDate
```

```
Jugador>>initializeConNombre: unString fechaNac: aDate  
    nombre:= unString.  
    fechaNacimiento := aDate.  
    zona := ZonaJugador new
```



```
Jugador>>newConNombre: unString fechaDeNacimiento: aDate zona: unaZona  
    ^super new initializeConNombre: unString fechaNac: aDate zona: unaZona
```

```
Jugador>>initializeConNombre: unString fechaNac: aDate zona: unaZona  
    nombre:= unString.  
    fechaNacimiento := aDate.  
    zona := unaZona
```

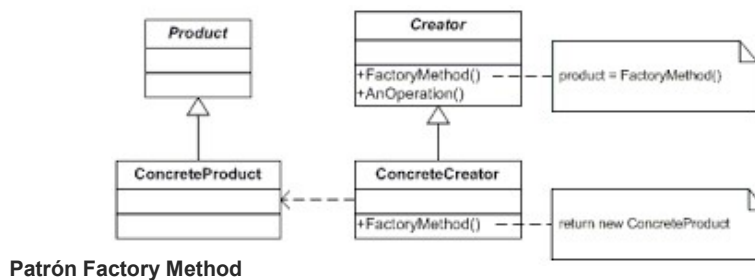
[Alternativa:]

- Como setear la estrategia en el contexto?
- Si no hay muchas combinaciones de Strategies y contextos, es una buena práctica 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 retorne una instancia del mismo con el strategy correspondiente, por cada subclase de Strategy.

```
Jugador>>newEnZonaAConNombre: unString fechaDeNacimiento: aDate  
    ^super new  
        initializeConNombre: unString  
        fechaNac: aDate  
        zona: ZonaA new
```

Aclaración

- Jugador>>newEnZonaAConNombre: fechaDeNacimiento: es un “factory” o “creation method”, pero no es el patrón Factory Method ! (como lo llama Fowler)



Patrón Factory Method

Alejandra Garrido - Objetos 2 - UNLP

25

Replace Cond. Logic w/ Strategy: Pros y Contras

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

26

Alejandra Garrido - Objetos 2 - UNLP

[

]

■ State o Strategy?

[

State o Strategy?

]

- 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

[State vs. Strategy]

- El estado es privado del objeto, ningún otro objeto sabe de él. vs.
- ≠ El Strategy suele setearse por el cliente, que debe conocer las posibles estrategias concretas.
- Cada State puede definir muchos mensajes. vs.
- ≠ Un Strategy suele tener un único mensaje público.
- Los states concretos se conocen entre si.
- ≠ Los strategies concretos no.

[Replace State-Altering Conditionals with State]

- Las expresiones condicionales que controlan las transiciones de estado de un objeto son complejas.



- *Reemplazar los condicionales con States que manejen estados específicos y transiciones entre ellos.*

Replace State-Altering Conditionals with State

■ 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

31

Alejandra Garrido - Objetos 2 - UNLP

Replace State-Altering Conds. with State. Mecánica

1. Si hay una sola v.i. que se compara con distintas constantes then *“Replace Type-Code with Class”*

```
Libro>>estáPrestado
    ^condicion = "Prestado"
Libro>>estáEnReencuadernación
    ^condicion = "EnReencuadernacion"
Libro>>estáReservado
    ^condicion = "Reservado"
Libro>>reservar
    condicion := "Reservado"
```

• Mecánica:

1. Aplicar “Self-Encapsulate Field [F]”. C & T.
2. Crear una nueva clase: superclase del State.
3. Agregar una v.i. en la clase contexto para el estado y su setter.
4. Cambiar los setters. C&T
5. Cambiar los getters. C&T
6. Borrar la vieja v.i.

32

Copyright Alejandra Garrido - LIFIA - UNLP

Replace State-Altering Conds. with State. Mecánica

1. Si hay más de una v.i. que mantiene el estado then *"Extract Class"*

Class Libro
instance variables:
 "prestado reservado
 enReencuadernacion
 ..."

- Mecánica:
 1. Crear una nueva clase: superclase del State.
 2. Agregar una v.i. en la clase contexto para el estado y su setter.
 3. Aplicar *"Move Field"*[F]. C&T
 4. Aplicar *"Move Method"*[F]. C&T

33

Copyright Alejandra Garrido - LIFIA - UNLP

Replace State-Altering Conds. with State. Mecánica

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. Dejarlos estos métodos como abstractos en la superclase o como métodos por defecto.

34

Copyright Alejandra Garrido - LIFIA - UNLP

Cómo cambia de estado el Contexto del State?

- El State tiene que conocer el Contexto o recibirlo como parámetro
- El Contexto define un método *changeState()*

Replace State-Altering Conditionals with State

- Beneficios y desventajas
 - + 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.
 - Complica el diseño cuando la lógica de transición de estados ya es fácil de seguir.

[Qué pasa con los tests?]

- Los tests no deberían limitarnos o restringirnos al momento de aplicar refactoring
- Hay estrategias, o mejor dicho, patrones para hacer los tests más resistentes, es decir, para evitar “Fragile Tests” (“Xunit Test Patterns”. Gerard Meszaros. Addison-Wesley 2007)
- 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)
SUT: System Under Test

Alejandra Garrido - Objetos 2 - UNLP

37

[Test patterns que solucionan “Fragile Test”]

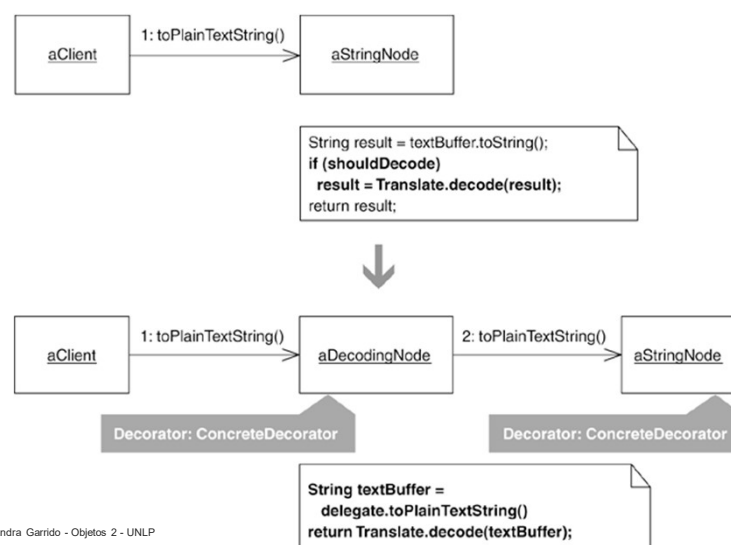
- Una de las causas de Fragile Test es “sensibilidad a la interface”, o “sensibilidad al protocolo” del SUT (en lenguajes tipados estáticamente, el test falla al compilar)
- *Test Utility Method*: permite encapsular la dependencia innecesaria con el API del SUT en un método, por ejemplo, de creación. Cuando 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)
- “Connection Between Safe Refactorings and Acceptance Test Driven Development”. Carlos Fontela y Alejandra Garrido. IEEE Latin America Transactions, Vol. 11, No. 5, Sept. 2013

Alejandra Garrido - Objetos 2 - UNLP

38

- Otros refactorings to patterns...

Move Embellishment to Decorator



[¿Por qué refactoring es importante?]

- Nuestra única defensa contra el deterioro del software.
- Es una técnica importante porque:
- Facilita la incorporación de código
- Permite *agregar patrones* después de haber escrito el programa; permite *transformar un programa en framework*.
- Permite preocuparse por la generalidad mañana; hoy solo hay que hacerlo andar
“*Make it work. Make it right. Make it fast*”. Kent Beck.
- Es decir, permite ser ágil en el desarrollo



41

Alejandra Garrido - Objetos 2 - UNLP

[Pagando la deuda técnica]

- Si consideramos los problemas de diseño como una deuda que se nos va acumulando...
- el refactoring nos permite **pagar la deuda**



Alejandra Garrido - Objetos 2 - UNLP

42

[Referencias]

- “Refactoring to Patterns”. Joshua Kerievsky. Addison Wesley 2005.
- “Design Patterns”. Gamma et al. Addison Wesley 1995.
- “Xunit Test Patterns”. Gerard Meszaros. Addison-Wesley 2007
- “Connection Between Safe Refactorings and Acceptance Test Driven Development”. Carlos Fontela y Alejandra Garrido. IEEE Latin America Transactions, Vol. 11, No. 5, Sept. 2013