Refactoring de código

Mejorar el diseño de código existente

Alejandra Garrido – Objetos 2

Organización catálogo Fowler

- Composición de métodos
- Mover aspectos entre objetos
- Organización de datos
- Simplificación de expresiones condicionales
- Simplificación en la invocación de métodos
- Manipulación de la generalización
- Big refactorings

Composición de métodos

- Permiten “distribuir” el código adecuadamente.
- Métodos largos son problemáticos
- Contienen:
  - mucha información
  - lógica compleja

- Extract Method
- Inline Method
- Replace Temp with Query
- Split Temporary Variable
- Replace Method with Method Object
- Substitute Algorithm

Nota: Extract Method en Date class>>readFrom:

Mover aspectos entre objetos

- Ayudan a mejorar la asignación de responsabilidades

- Move Method
- Move Field
- Extract class
- Inline Class
- Remove Middle Man
- Hide Delegate

Organización de datos

- Facilitan la organización de atributos

- Self Encapsulate Field
- Encapsulate Field / Collection
- Replace Data Value with Object
- Replace Array with Object
- Replace Magic Number with Symbolic Constant

Simplificación de expresiones condicionales

- Ayudan a simplificar los condicionales

- Decompose Conditional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Replace Conditional with Polimorfism

Nota: Decompose Conditional en StandardRoom

Simplificación de invocación de métodos

- Sirven para mejorar la interfaz de una clase

- Rename Method
- Preserve Whole Object
- Introduce Parameter Object
- Parameterize Method

Manipulación de la generalización

- Ayudan a mejorar las jerarquías de clases

- Push Up / Down Field
- Push Up / Down Method
- Pull Up Constructor Body
- Extract Subclass / Superclass
- Collapse Hierarchy
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance

Nota: Push Up Method en Employee

Refactoring to patterns

Mejoras de diseño complejas

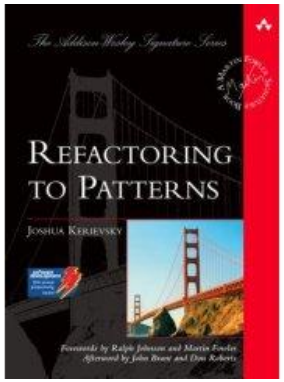
Relación entre patrones y refactorings

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

La panacea de los patrones

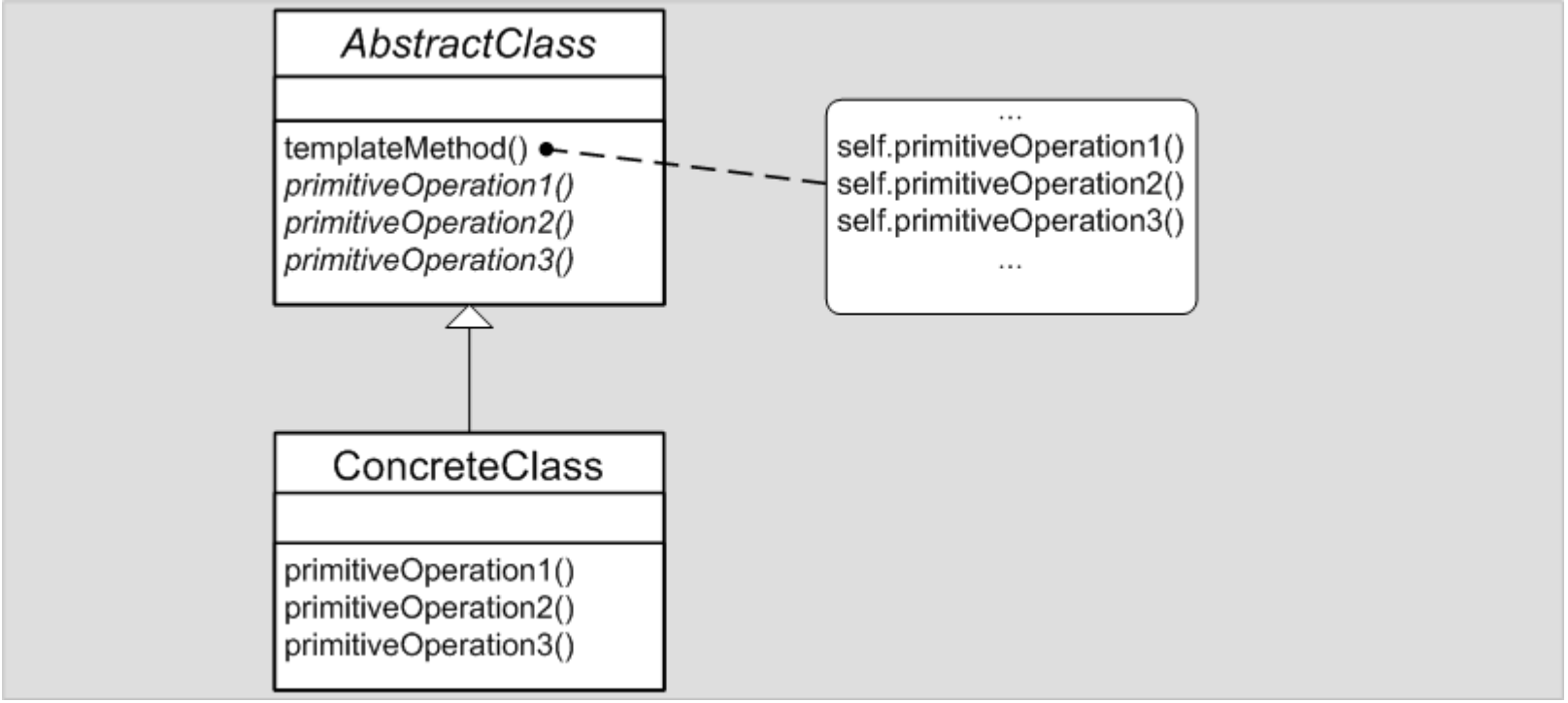
- Los patrones son tentadores para no quedarnos envueltos y arrastrar un mal diseño.

- También nos pueden llevar al otro extremo. Por esto es muy importante conocer las consecuencias tanto positivas como negativas de un patrón.

Refactoring to Patterns

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

Patrón Template Method



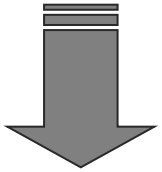
Propósito de un patrón vs. Problema que soluciona

- 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.*

Refactoring “Form Template Method”

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

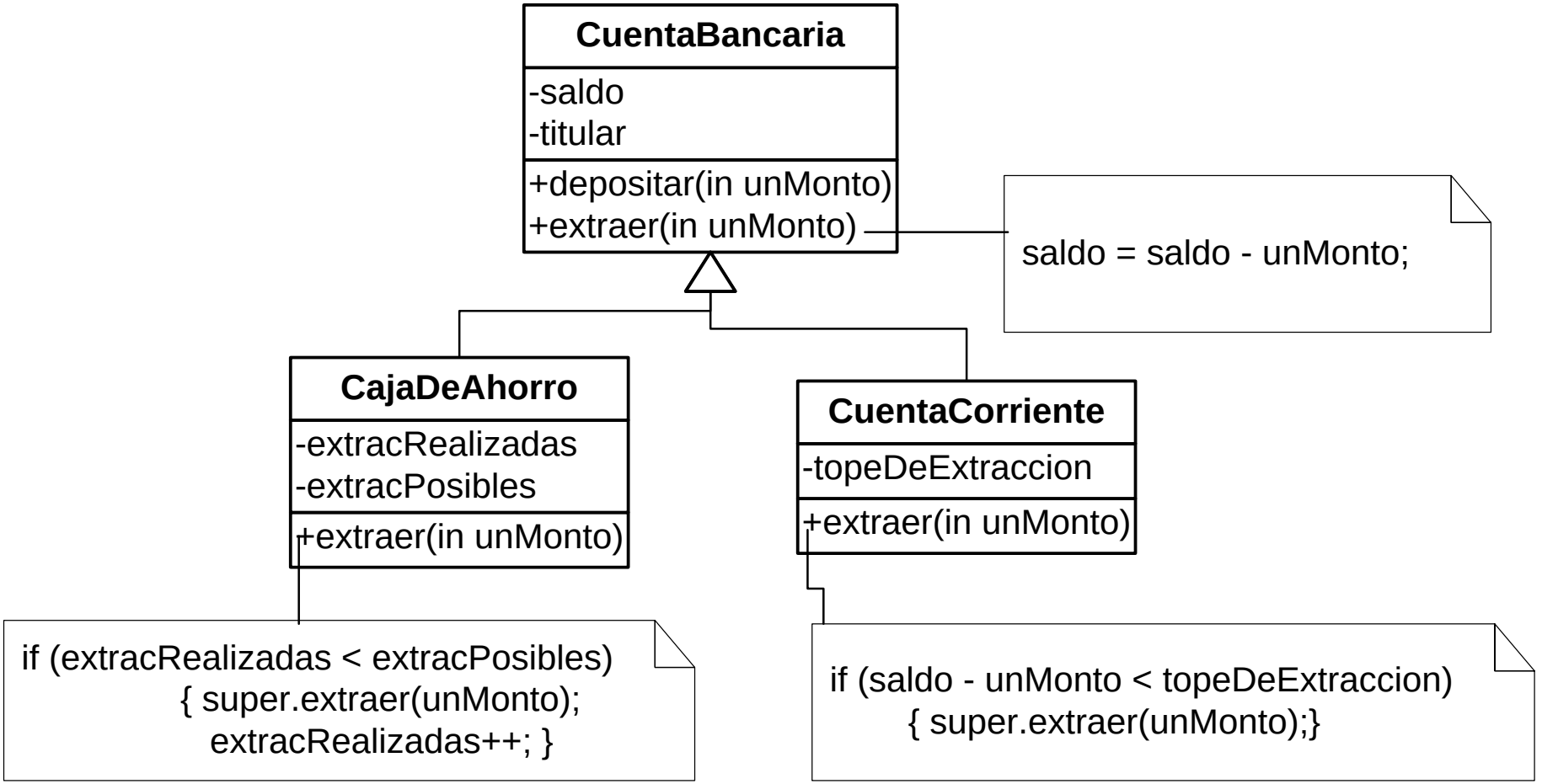


- Generalizar los métodos extrayendo sus pasos en métodos de la misma signatura, y luego subir a la superclase común el método generalizado para formar un Template Method.

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

Aplicar Template Method



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

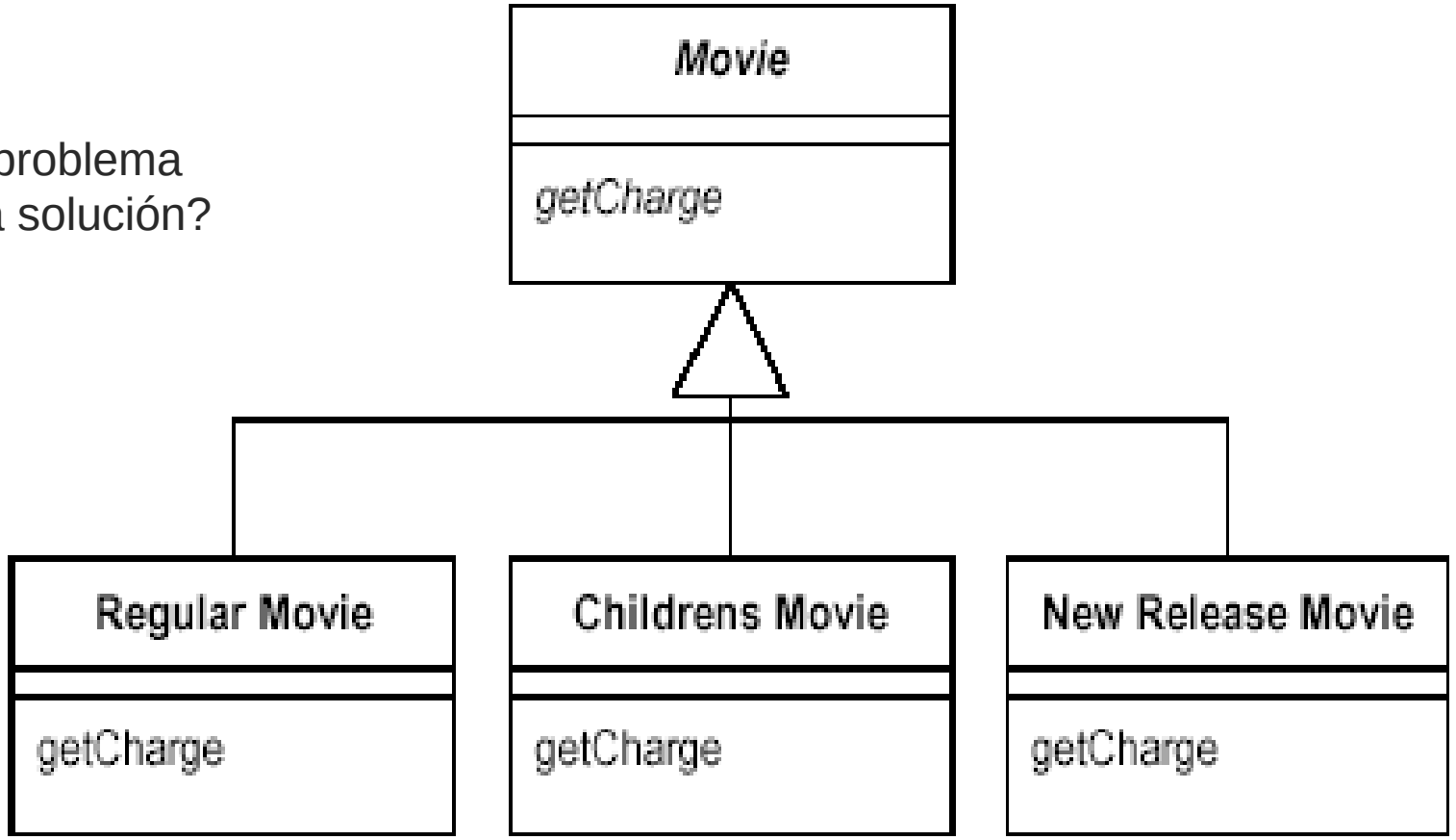
Volvemos al video club

```
Class Movie…
abstract double
    getCharge(int daysRented);
```

```
Class RegularMovie
double getCharge(int daysRented){
    double result = 2;
    if (daysRented > 2) result += (daysRented - 2) * 1.5;
    return result;
}
Class ChildrensMovie
double getCharge(int daysRented){
    double result = 1.5;
    if (daysRented > 3) result += (daysRented - 3) * 1.5;
    return result;}

Class NewReleaseMovie
double getCharge(int daysRented){
     return daysRented * 3;}
```

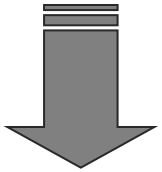
Después de “Replace conditional with polimorphism”

Algún problema con esta solución?



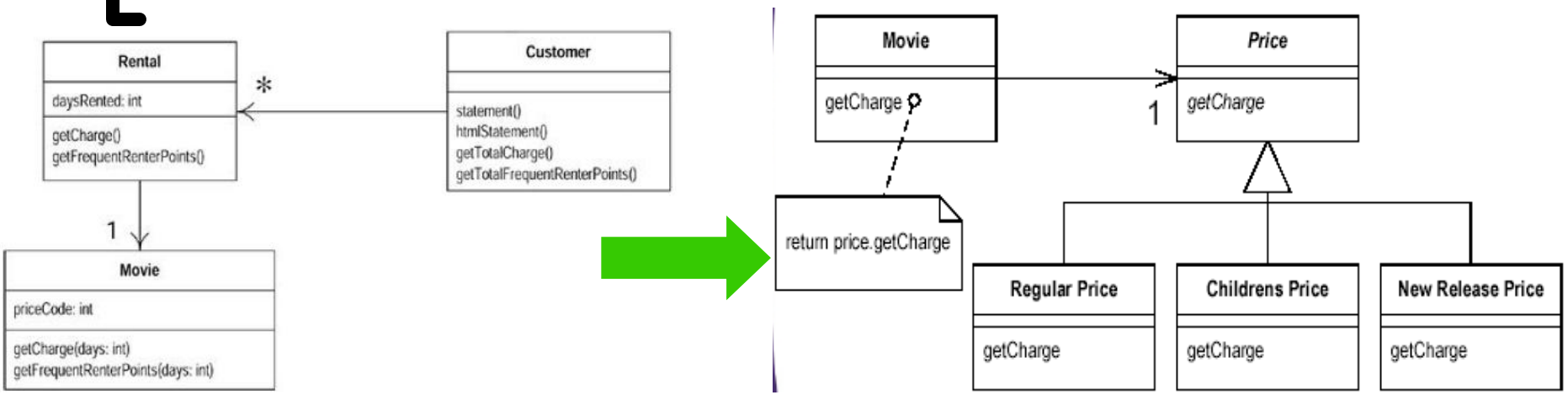
Replace Conditional Logic with Strategy

- 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

Replace Cond. Logic w/ Strategy



1. Moveremos el código con los condicionales a Price usando **Move Method**.
2. Luego usaremos **Extract Parameter** para poder setearle un Price a Movie.
3. Por último, usaremos el **Replace Conditional with Polymorphism** en Price para eliminar la sentencia switch

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 strategias y contextos.

Método getCharge() en la clase Movie

```
class Rental...
   double getCharge() {
      return _movie.getCharge(_daysRented);
}
```

```
class Movie …
   double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
        return result;
   }
```

Reimplementando el getCharge() en cada subclase

Class **Movie**…
```
private Price _price;

public Movie(String name, Price price)
{  _name = name;
   _price = price;
}

double getCharge(int daysRented) {
   return _price.getCharge(daysRented);
}

public void setPriceType(Price price) {
   _price = price;          }
```

Class **Price…**

```
abstract double getCharge(int
                          daysRented);
```

Class **RegularPrice**
```
double getCharge(int daysRented){
     double result = 2;
     if (daysRented > 2)
        result += (daysRented - 2) * 1.5;
     return result;
}
```

Class **ChildrensPrice**
```
double getCharge(int daysRented){
     double result = 1.5;
     if (daysRented > 3)
        result += (daysRented - 3) * 1.5;
     return result;}
```

Class **NewReleasePrice**
```
double getCharge(int daysRented){
      return daysRented * 3;}
```

Replace Cond. Logic w/ Strategy: Pros y Contras

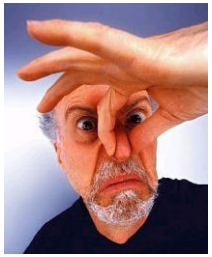
- Clarifica los algoritmos al reducir o remover la lógica condicional.
- Simplifica una clase moviendo variaciones de un algoritmo a una jerarquía separada
- Permite reemplazar un algoritmo por otro en runtime.
- Complica el diseño cuando se podría solucionar con subclases o simplificando los condicionales.

Ejercicio: biblioteca

- Supongamos una biblioteca que presta libros.
- De los libros se conoce: título, autor, isbn
- Para pedir un libro prestado hay que ser socio. De los socios se conoce:
  - nombre
  - numero de socio
  - direccion
  - telefono
  - libros en su poder

Ejercicio: biblioteca

```
Biblioteca>>prestarUnLibro: unLibro alSocio: unSocio
        unSocio pedirPrestado: unLibro

Socio>>pedirPrestado: unLibro
        (self puedeRetirarLibro)
        ifTrue: [
                (libro estáPrestado)
                ifFalse: [
                    libro prestarse.
                    self addLibro: unLibro]]
```



Supongamos ahora que los libros también pueden reservarse. Cómo cambia el código anterior?

Ejercicio: biblioteca

```
Biblioteca>>prestarUnLibro: unLibro alSocio: unSocio
        (unSocio puedeRetirarLibro)
        ifTrue: [unLibro prestarseA: unSocio]

Libro>>prestarseA: unSocio
        (self estáPrestado)
        ifFalse: [self prestarse.
                unSocio retirarLibro: self]
```

Ejercicio: biblioteca

Ahora los libros también pueden reservarse.

```
Libro>>prestarseA: unSocio
        (self estáPrestado)
        ifFalse:
            [(self estáReservado not or:
                [self reservadoA: unSocio])
                ifTrue:
                    [self prestarse.
                     unSocio retirarLibro: self]
```

Ejercicio: biblioteca

```
Libro>>reservarseA: unSocio
        (self prestado)
        ifFalse:
                [self prestarseA: unSocio]
        ifTrue:
                [self reservado
                   ifFalse: [self reservar: unSocio]]
```

Ejercicio: biblioteca

Ahora los libros también pueden enviarse a reencuadernación.

```
Libro>>prestarseA: unSocio
        (self estáPrestado or:
           [self estáEnReencuadernación])
        ifFalse:
                [(self estáReservado not or:
                    [self reservadoA: unSocio])
                        ifTrue:
                                [self prestarse.
                                 unSocio retirarLibro: self]
```

Ejercicio: biblioteca

```
Libro>>reservarseA: unSocio
        (self estáPrestado)
        ifFalse:
                [self estáEnReencuadernacion
                  ifFalse: [self prestarseA: unSocio]]
                  ifTrue:
                        [self estáReservado
                            ifFalse:[self reservar: unSocio]]
        ifTrue:
                [self estáReservado
                   ifFalse: [self reservar: unSocio]]
```

Situación

- El libro puede estar en diferentes estados y en función de eso reacciona
- La solución actual está llena de condicionales y agregar un nuevo estado implica cambiar todos los métodos del libro
- Cuál sería una mejor solución?

El patrón State

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

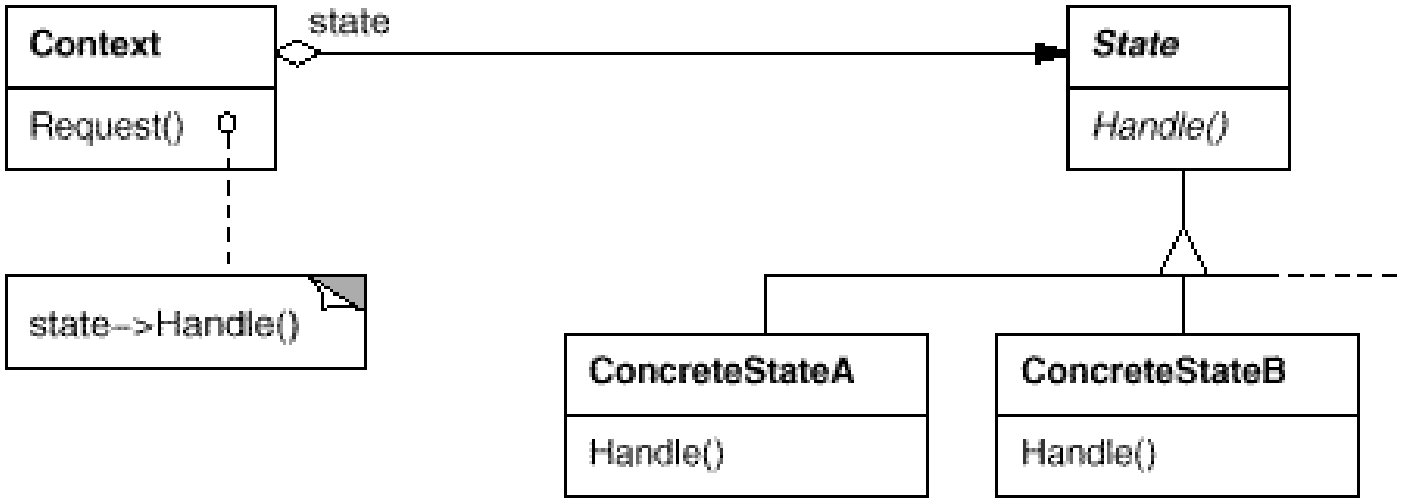
Patrón State. Aplicabilidad

- 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 y en muchas operaciones aparece el mismo condicional. El patron 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).

Patrón State. Estructura

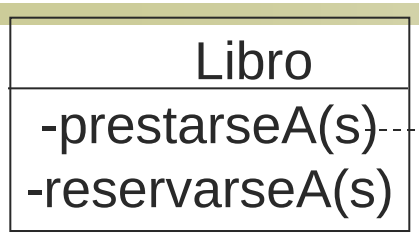


Patrón State. Consecuencias

+ Localiza el comportamiento relacionado con cada estado.
+ El State es transparente para las clases clientes.
+ Las transiciones entre estados son explícitas.
+ En el caso que los estados no tengan variables de instancia pueden ser compartidos.

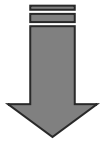
- Los estados suelen estar acoplados entre sí

Refactoring!

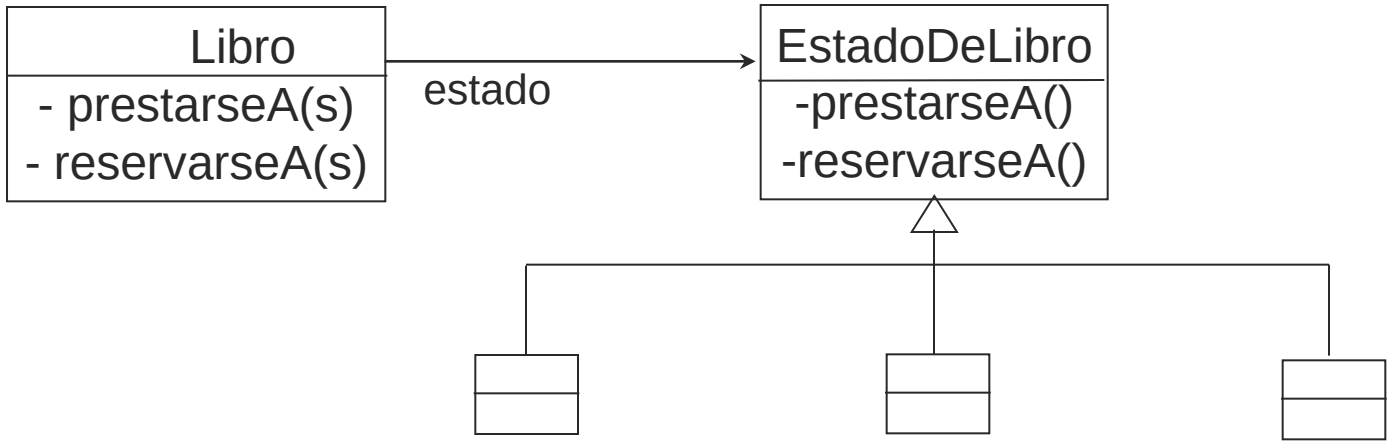



```
prestarseA(unSocio) {
        if (....)
          then
           { .... }
        else if (...)
              then
                    {...}
              else
                    {...}
```

Tests: Pass

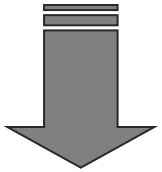


Replace State-Altering Conditional with State



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

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.

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.

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

Replace State-Altering Conds. with State. Mecánica

2. Crear una subclase del State por cada uno de los estados en los que la clase contexto puede entrar.

```
Libro>>reservarseA: unSocio
        (self estáPrestado)
        ifFalse:
                [self estáEnReencuadernacion
                  ifFalse: [self prestarseA: unSocio]]
                  ifTrue:
                        [self estáReservado
                           ifFalse:[self reservar: unSocio]]
        ifTrue:
                [self estáReservado
                  ifFalse: [self reservar: unSocio]]
```

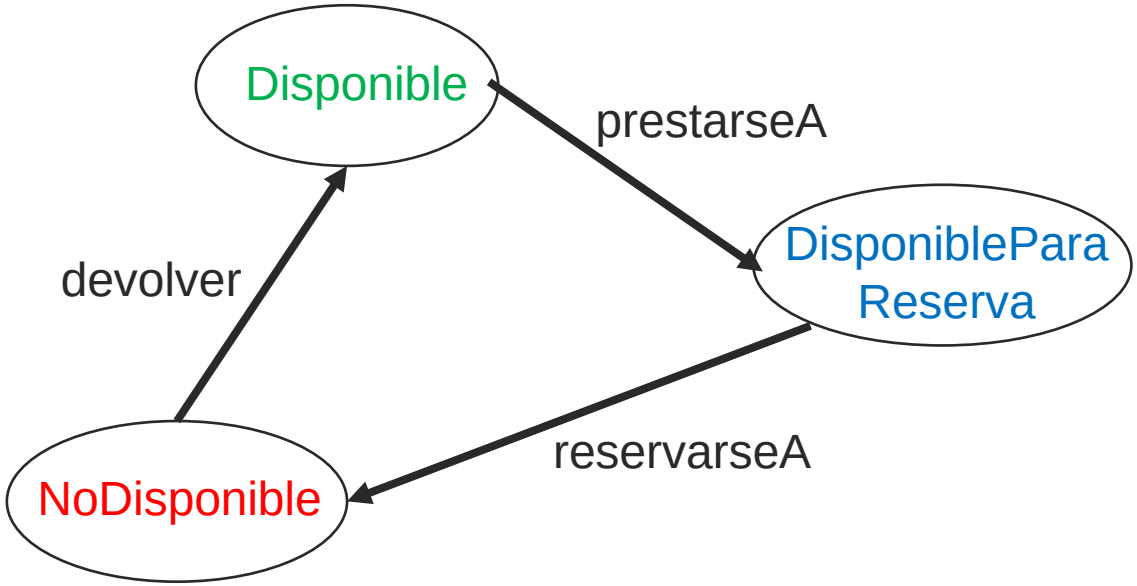
Disponible

DisponiblePara Reserva

DisponiblePara Reserva

NoDisponible

Diagrama de transición de estados



Replace State-Altering Conds. with State. Mecánica

3. Por cada método de la clase contexto con condicionales que cambiar el valor del estado, aplicar “Move Method” hacia la superclase de State.

```
EstadoDeLibro>>reservarseA: unSocio
        (self estáPrestado)
        ifFalse:
                [self estáEnReencuadernacion
                  ifFalse: [self prestarseA: unSocio]]
                  ifTrue:
                          [self estáReservado
                             ifFalse:[self reservar: unSocio]]
        ifTrue:
                [self estáReservado
                  ifFalse: [self reservar: unSocio]]
```

Replace State-Altering Conds. with State. Mecánica

4. Por cada estado concreto, copiar de la superclase los métodos que producen una transición desde ese estado. Sacar la lógica de comprobación que ya no hace falta.

```
LibroDisponibleParaReserva>>reservarseA: unSocio
        (self estáPrestado)
        ifFalse:
                [self estáEnReencuadernacion
                  ifFalse: [self prestarseA: unSocio]]
                ifTrue:
                        [self estáReservado
                          ifFalse:[self reservarseA: unSocio]]
        ifTrue:
                [self estáReservado
                  ifFalse: [self reservarseA: unSocio]]
```

Replace State-Altering Conds. with State. Mecánica

5. Borrar el cuerpo de estos métodos en la superclase y dejarles el cuerpo vacio (Kerievksy).
   O dejarlos como métodos abstractos (Fowler).

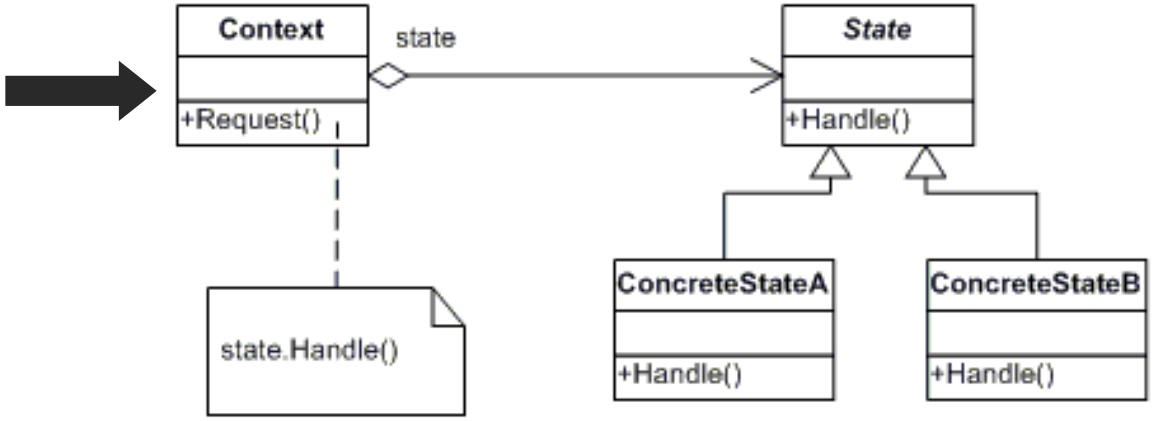
```
EstadoDeLibro>>reservarseA: unSocio
  ^self subclassResponsibility

EstadoDeLibro>>prestarseA: unSocio
  ^self subclassResponsibility
```

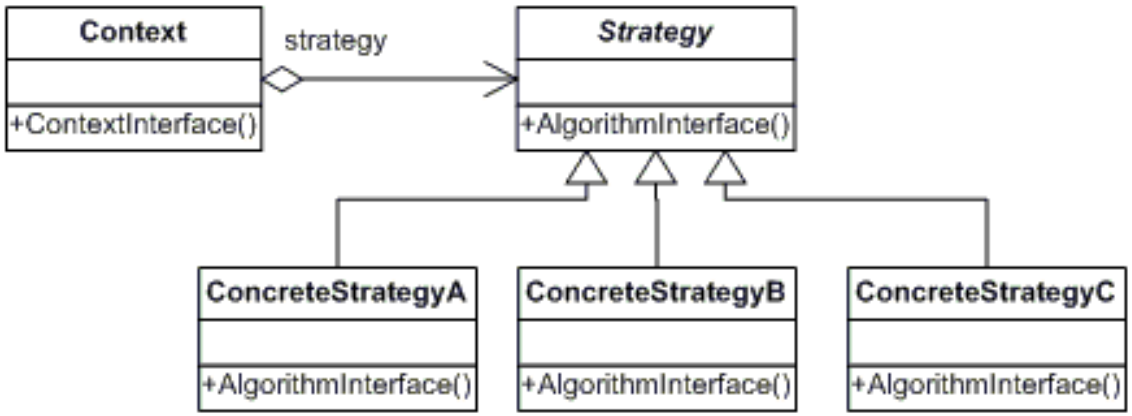
Haciendo memoria…

- En qué se asemejan y en qué se diferencian State de Strategy??

State vs. Strategy

State





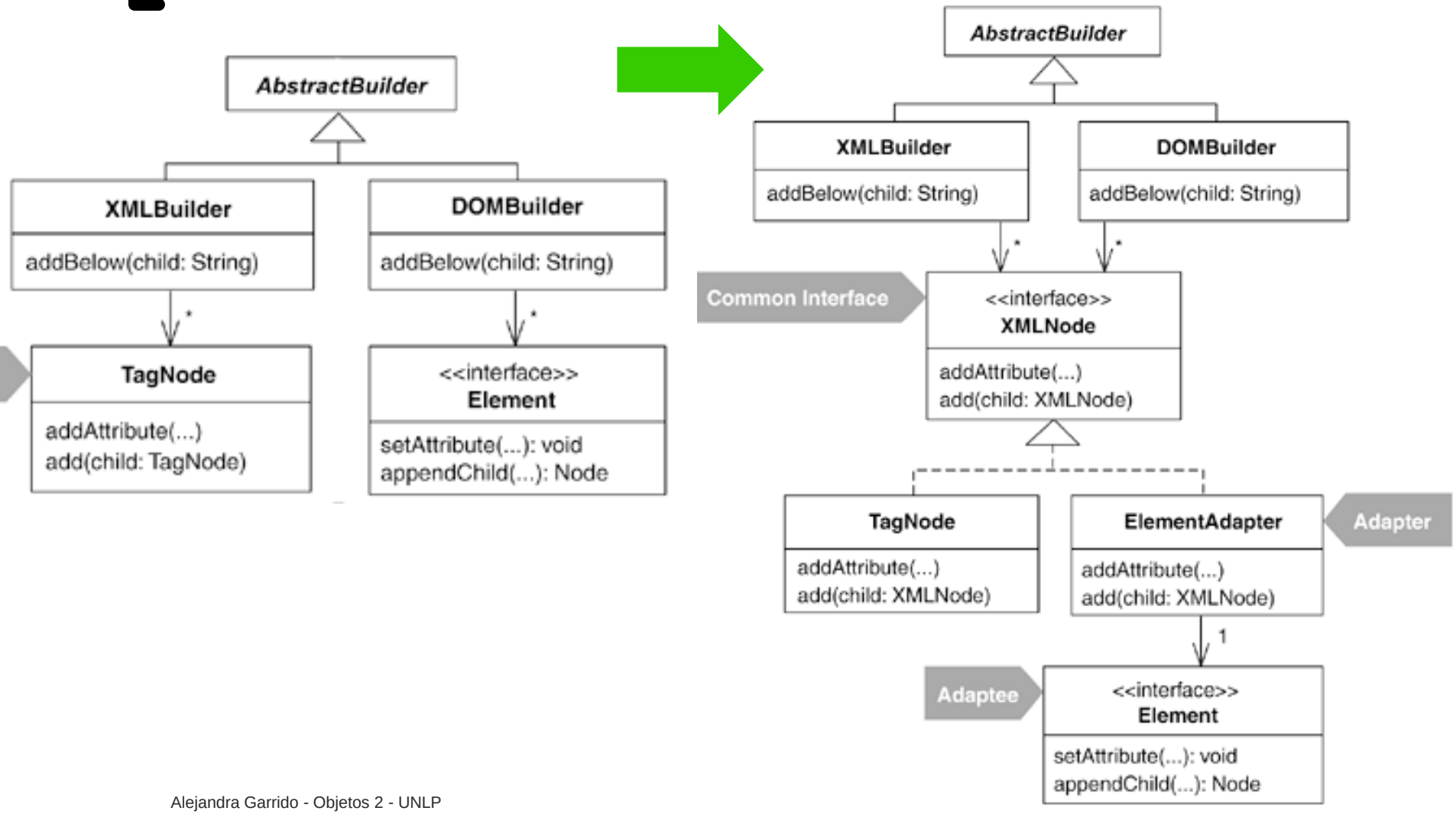
Strategy

State vs. Strategy

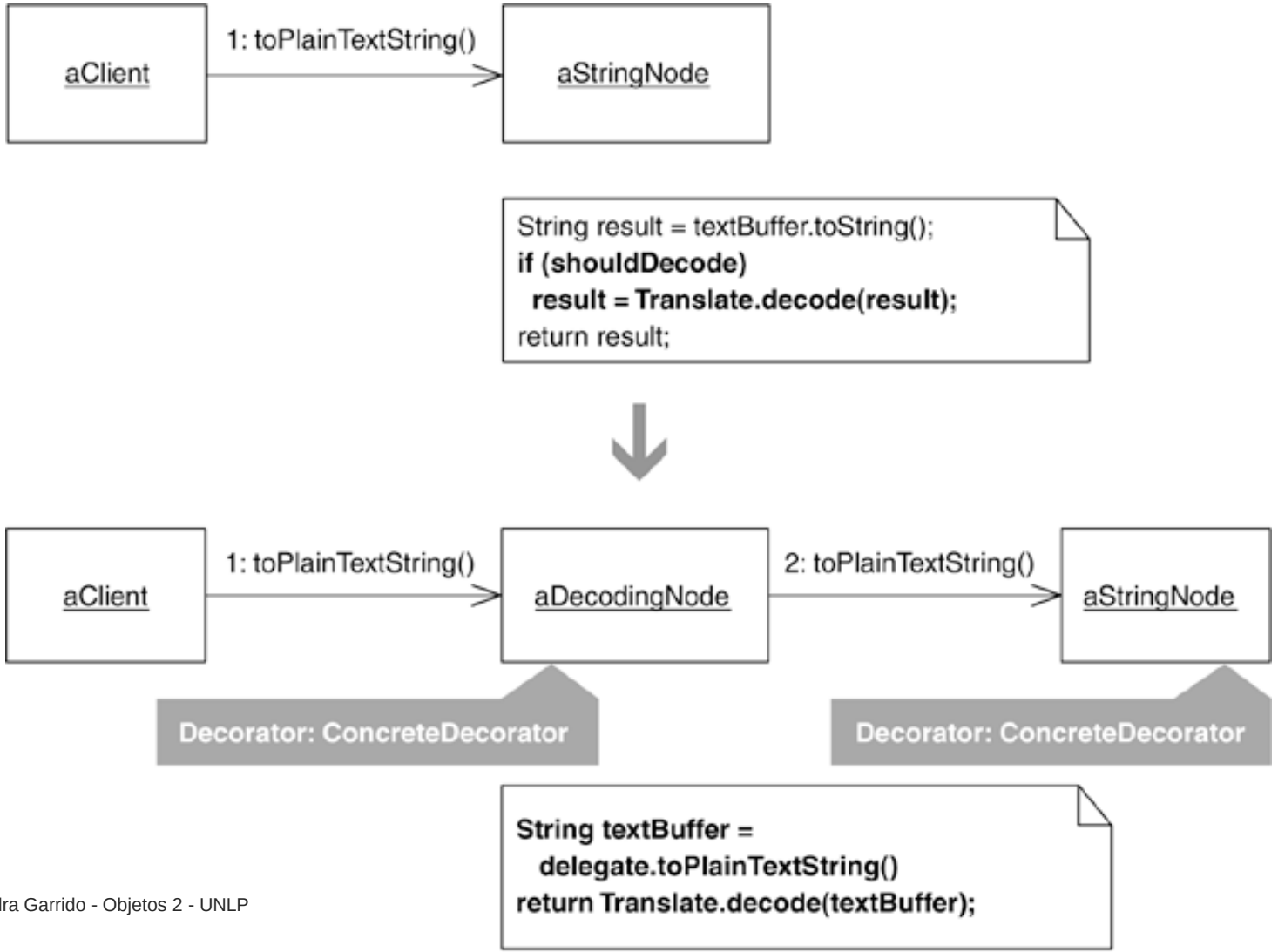
- Mismo diagrama de clases.
- Misma idea de delegación.
- Pero
  - El estado es privado del objeto, ningún otro objeto sabe de él. vs.
  - ≠ El Strategy suele setearse por el cliente, que debe conocer los posibles strategies concretos.
  - Un State define una máquina de estados con sus transiciones.
  - 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.

- Otros refactorings to patterns…

Unify Interfaces with Adapter



Move Embellishment to Decorator



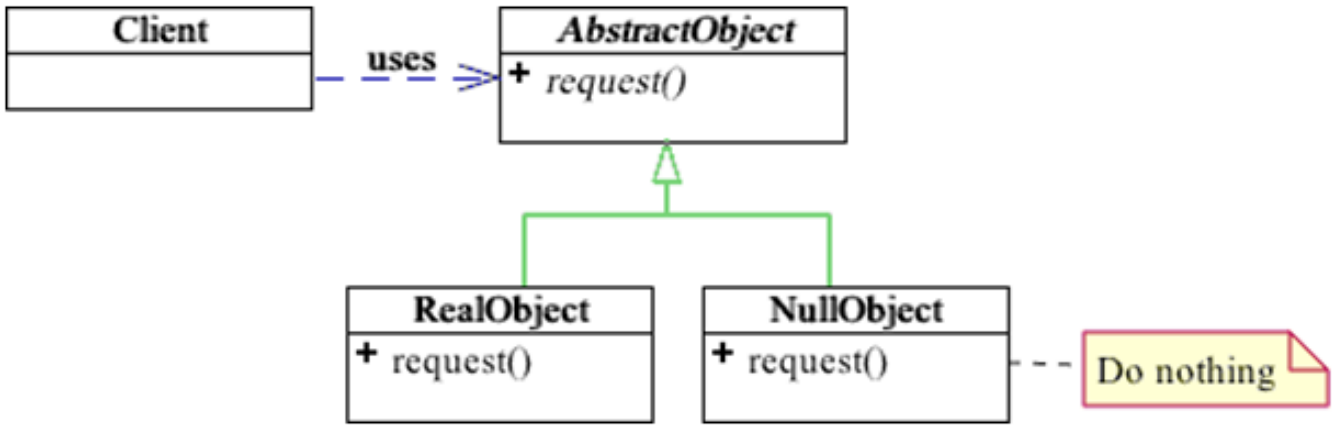
Ejercicio con interfaces

- Algunos elementos de la interface son clickeables, aceptan doble-click, drag-drop, otros nada.

```
class NavigationApplet
boolean mouseDown(…)
  if (mouseEventHandler != null)
        return mouseEventHandler.mouseDown(…)
  return true;

boolean mouseMove(…)
  if (mouseEventHandler != null)
        return mouseEventHandler.mouseMove(…)
  return true;

class MainMenuApplet
//idem
```

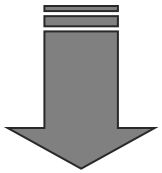
Patrón Null Object

- A Null Object provides a surrogate for another object that shares the same interface, but does nothing.



Refactoring Introduce Null Object

- La lógica para manipular un atributo o variable nula está duplicada a través del código.



- *Reemplazar esa lógica con un Null Object, un objeto que provee el comportamiento nulo apropiado.*

Introduce Null Object. Mecánica

1. Extract Subclass [Fowler]. C
2. Buscar un chequeo por null en el código de la clase cliente. Redefinir el método en el null object para que implemente el comportamiento alternativo. C y repetir para todos los null check
3. Encontrar una clase cliente que contenga alguna ocurrencia del null check e inicializar la var. a la que se refiere con un null object, lo antes posible. C
4. En la clase del paso 3, sacar los null checks. C & T
5. Repetir 3 y 4 por cada clase.
6. Usar Singleton para crear una sola instancia del null object

Introduce Null Object

- Algunos elementos de la interface son clickeables, aceptan doble-click, drag-drop, otros nada.

```
class NavigationApplet
boolean mouseDown(…)
  return mouseEventHandler.mouseDown(…)
boolean mouseMove(…)
  return mouseEventHandler.mouseMove(…)
```

```
class NullMouseEventHandler extends MouseEventHandler
boolean mouseDown(…)
  return true;
boolean mouseDown(…)
  return true;
```

Introduce Null Object: Pros y Cons

+ Previene errores por null sin duplicar la lógica del null.
+ Simplifica el código minimizando los testeos por null.
– Complica el diseño si el sistema necesita pocos testeos por null.
– Complica el mantenimiento: los Null Objects deben redefinir todos los métodos públicos heredados de la superclase.

Referencias

- “Refactoring to Patterns”. Joshua Kerievsky. Addison Wesley 2005.
- “Design Patterns”. Gamma et al. 1995.
- Null Object Pattern: http://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP2002/2002_Henney_NullObject.pdf
- Null Object Pattern: http://www.oodesign.com/null-object-pattern.html