Refactoring to patterns

Mejoras de diseño complejas

Cómo ayuda el refactoring?

- Una vez que tengo código que funciona y pasa los tests
- Provee mecanismos que solucionan problemas de diseño
- A través de cambios ***pequeños***
  - Hacer muchos cambios pequeños es más fácil y más seguro que un gran cambio
  - Me permite testear después de cada cambio
  - Cada pequeño cambio pone en evidencia otros cambios necesarios

Hacia cambios más complejos

- Una secuencia de refactorings me puede llevar a aplicar un gran cambio
  - *Don Roberts*. Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana Champaign, 1999.
  - *Michael Feathers*. Working Effectively with Legacy Code. Prentice Hall. 2005.
  - *Joshua Kerievsky.* Refactoring to Patterns. Addison-Wesley. 2005.

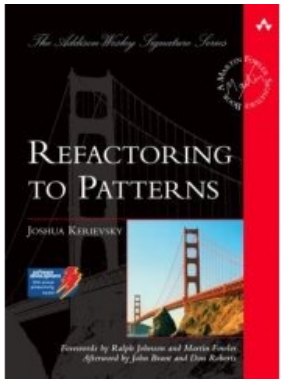
Relación entre patrones y refactorings

- "Design patterns provide targets for your refactorings." (GOF, 1995)

- "Patterns are where you want to be; refactorings are ways to get there from somewhere else." (Fowler, 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?

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
    (pero a la larga, el encanto de los patrones puede hacer que perdamos de vista formas más simples de escribir código).
- Consecuencias:
  - El código sofisticado, complejo, se queda y complica el mantenimiento.
  - Nadie lo entiende. Nadie lo quiere tocar.
  - Como otros no lo entienden generan copias, código duplicado.

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.

- Introducimos patrones sólo cuando se necesitan

Refactoring to Patterns

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

En la clase pasada vimos...

Malos olores

- Código duplicado
  - Extract Method
  - Pull Up Method
  - Form Template Method

Soluciones a Código duplicado

- Refactoring Extract Method:
  - lo aplicamos para extraer el código que mostraba los resultados de un jugador y pudimos llamarlo 2 veces para cada jugadores, eliminando la duplicación
- Refactoring Pull Up Method:
  - cuando aparece código duplicado en métodos de las subclases, se mueve el método a la superclase

Supongamos que la clase pasada hubierámos llegado a esto

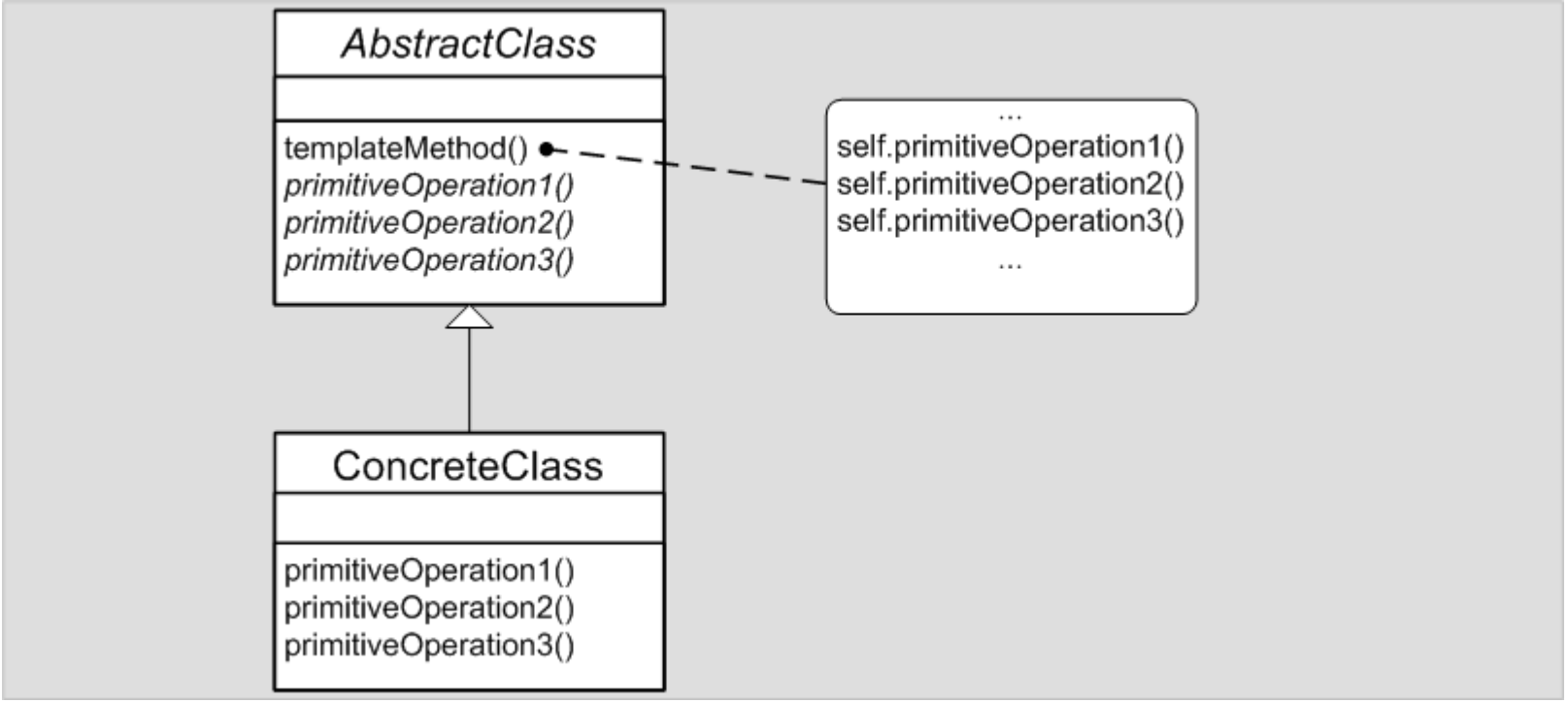
```
public class JugadorZonaA {
 String puntosEnPartidoToString(Partido partido) {
   String result = "Puntaje del jugador: " + nombre() + ": ";
   for (int gamesGanados: partido.puntosPorSetDe(this))
       result += Integer.toString(gamesGanados) + ";";
   result += "Puntos del partido: ";
   result += Integer.toString(totalGamesEnPartido(partido)*2);
   return result;   }

public class JugadorZonaB {
 String puntosEnPartidoToString(Partido partido) {
   String result = "Puntaje del jugador: " + nombre() + ": ";
   for (int gamesGanados: partido.puntosPorSetDe(this))
       result += Integer.toString(gamesGanados) + ";";
   result += "Puntos del partido: ";
   result += Integer.toString(totalGamesEnPartido(partido));
   return result;   }
}
```

La única diferencia

Qué más podemos hacer?

- Sigue habiendo código duplicado en las subclases.
- Todos los jugadores van a mostrar sus puntos de forma similar, solo cambia la última línea
- Qué patrón puedo usar para solucionarlo?

Patrón Template Method



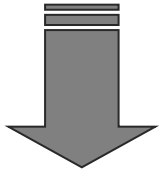
Propósito de un patrón vs. Problema que soluciona

- 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.*

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

Aplicamos Form Template Method

- Cuál es el método similar?
- Cuáles son los métodos idénticos?
- Cuáles son los métodos únicos?

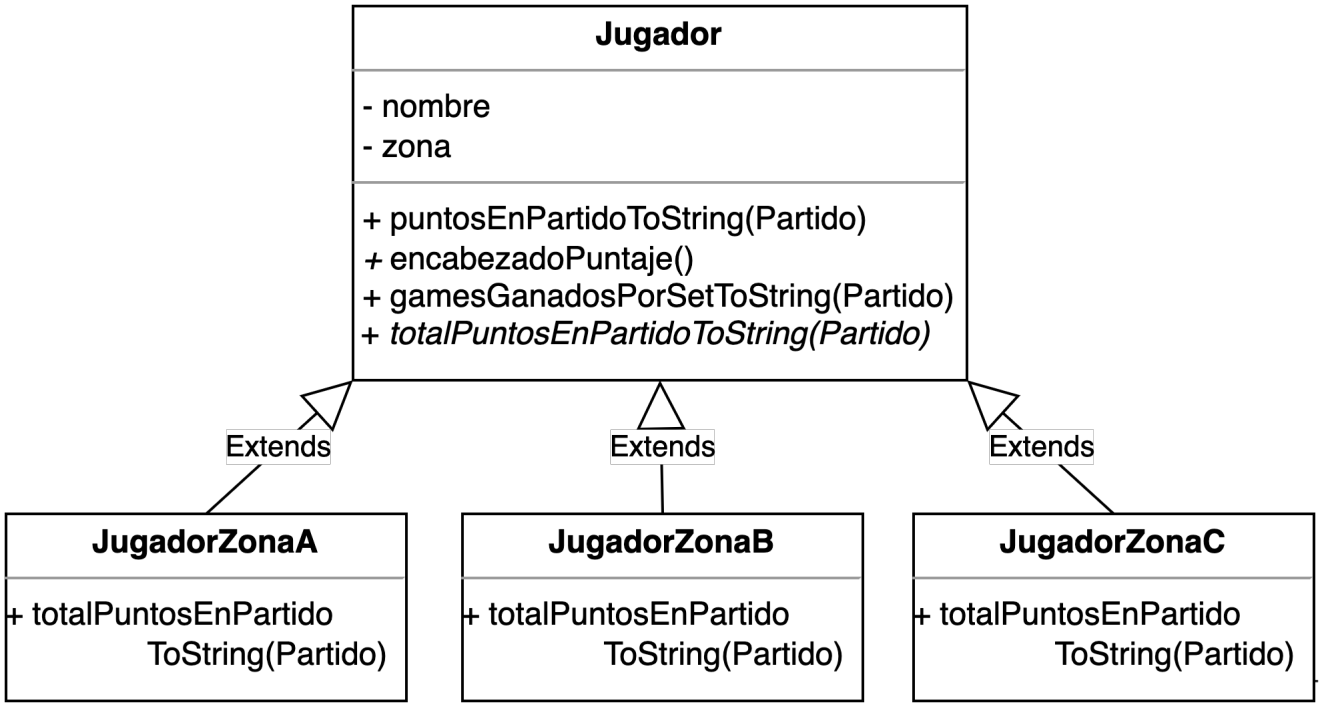
Método similar

```
public class JugadorZonaA {
 String puntosEnPartidoToString(Partido partido) {
   String result = "Puntaje del jugador: " + nombre() + ": ";
   for (int gamesGanados: partido.puntosPorSetDe(this))
       result += Integer.toString(gamesGanados) + ";";
   result += "Puntos del partido: ";
   result += Integer.toString(totalGamesEnPartido(partido)*2);
   return result;   }

public class JugadorZonaB {
 String puntosEnPartidoToString(Partido partido) {
   String result = "Puntaje del jugador: " + nombre() + ": ";
   for (int gamesGanados: partido.puntosPorSetDe(this))
       result += Integer.toString(gamesGanados) + ";";
   result += "Puntos del partido: ";
   result += Integer.toString(totalGamesEnPartido(partido));
   return result;   }
}
```

Se extraen en métodos únicos

Después de Form Template Method

```
public class Jugador {
 String puntosEnPartidoToString(Partido partido) {
   return (this.encabezadoPuntaje() +
             this.gamesGanadosPorSetToString(partido) +
             this.totalPuntosEnPartidoToString(partido));
```



Form Template Method: Pros y Contras

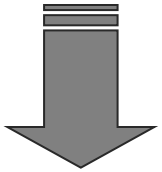
- Elimina código duplicado en las subclases moviendo el comportamiento invariante a la superclase.
- Simplifica y comunica efectivamente los pasos de un algoritmo genérico
- Permite que las subclases adapten fácilmente un algoritmo
- Complica el diseño cuando las subclases deben implementar muchos métodos para sustanciar el algoritmo

Otro requerimiento

- Supongamos que antes de hacer subclases de Jugador hubiéramos sabido que un jugador puede cambiar de zona

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 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 referenciar al strategy y un setter (generalmente el constructor del contexto)
    2) Dejar un método en el contexto que delegue
    3) Elegir los parámetros necesarios para pasar al strategy (el contexto entero? Sólo algunas variables? Y en qué momento?)
    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.

1) Crear superclase Strategy

```
public Jugador(String unNombre) {
        nombre = unNombre;
        zona = new ZonaJugadorStrategy();
}
public String puntosEnPartidoToString(Partido partido) {
// ...
    if (zona == "A")
        result += Integer.toString(totalGames * 2);
    if (zona == "B")
        result += Integer.toString(totalGames);
    if (zona == "C")
            if (partido.ganador() == this)
                    result += Integer.toString(totalGames);
            else
                    result += Integer.toString(0);
    //...
```

Ajustar antes de mover

```
public String puntosEnPartidoToString(Partido partido) {
// ...
    result += Integer.toString(puntosGanadosEnPartido(partido));
    return result;
}

public int puntosGanadosEnPartido(Partido partido) {
    if (zona == "A")
        return (totalGamesEnPartido(partido) * 2);
    if (zona == "B")
        return totalGamesEnPartido(partido);
    if (zona == "C")
        if (partido.ganador() == this)
                return totalGamesEnPartido(partido);
        else
                return 0;
}
```

2) Move Method

- Cuáles son los parámetros necesarios para pasar al strategy? (el contexto entero? Sólo algunas variables?)
- Y en qué momento se pasan esos parámetros?

3) Extract Parameter

```
public class Jugador {
 public Jugador(String unNombre) {
        nombre = unNombre;
        zona = new ZonaJugador(this);
 }
```

```
public class Jugador {
 public Jugador(String unNombre, ZonaJugador unaZona) {
        nombre = unNombre;
        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.
  - **Encapsulate Classes with Factory [Kerievsky]**: definir un método en el contexto que retorne una instancia del mismo con el strategy correspondiente, por cada subclase de Strategy.

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.

- 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

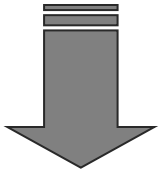
- El estado es privado del objeto, ningún otro objeto sabe de él. vs.
- ≠ El Strategy suele setearse por el cliente, que debe conocer las posibles estrategias concretas.

- 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

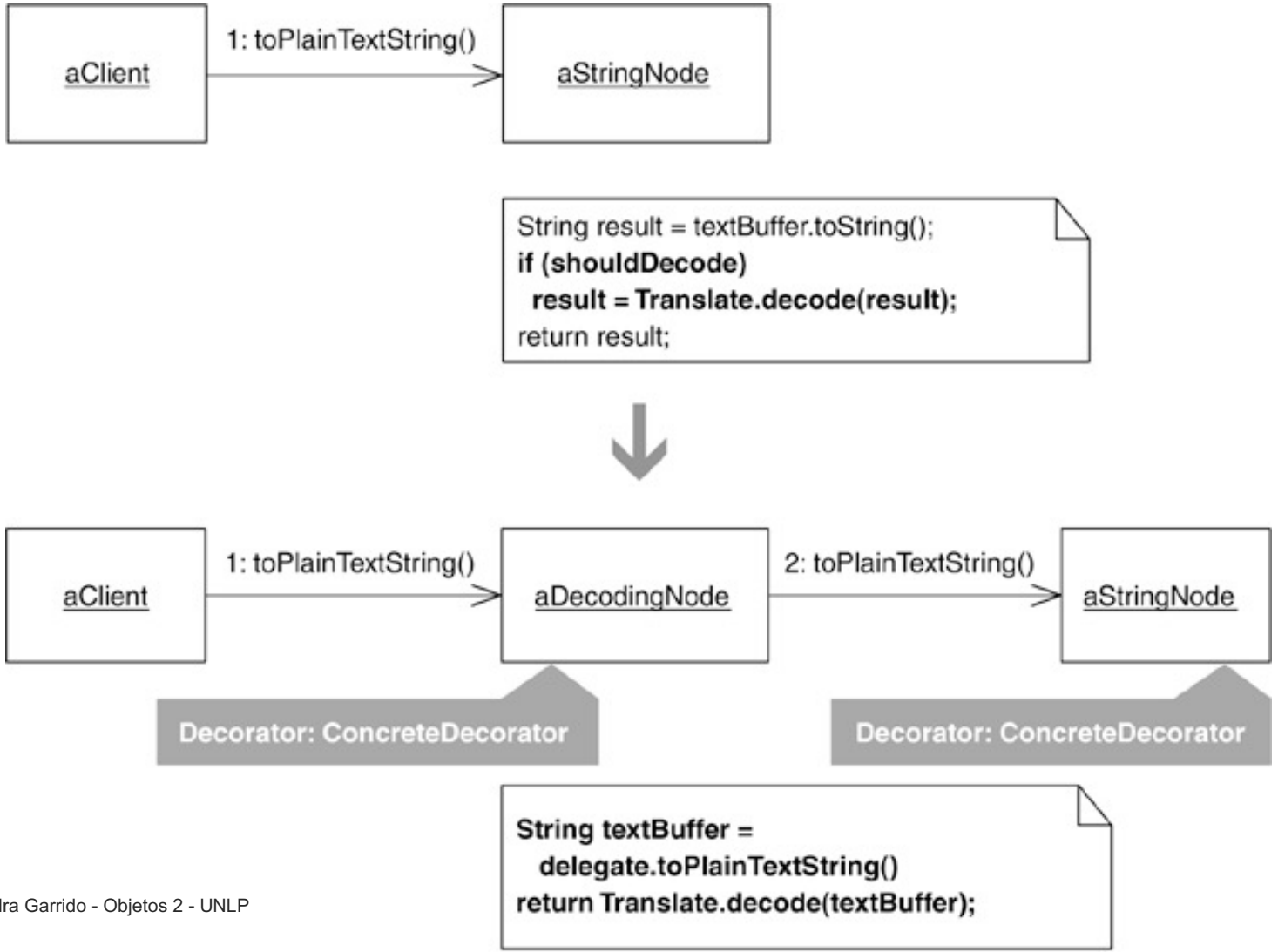
Replace State-Altering Conds. with State. Mecánica

1. Aplicar *"Replace Type-Code with Class"* para crear una clase que será la superclase del State a partir de la v.i. que mantiene el estado
2. Aplicar "*Extract Subclass"* [F] para crear una subclase del State por cada uno de los estados de la clase contexto.
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.

Replace State-Altering Conditionals with State

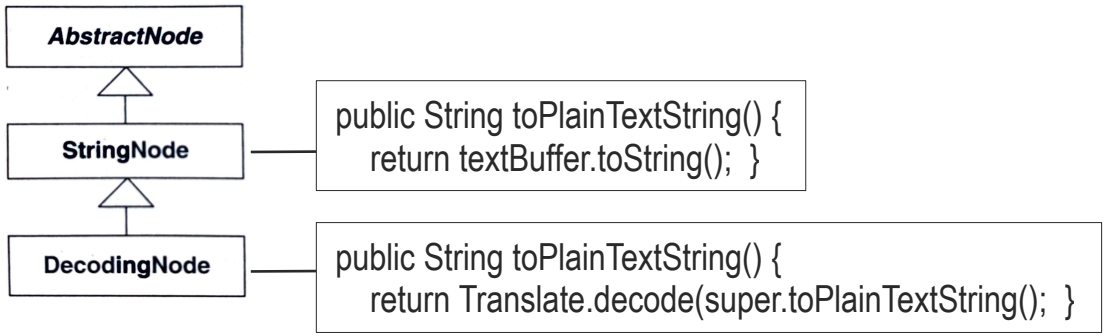
- Beneficios y desventajas
  - + Reduce o remueve la lógica condicional de cambio de estado.
  - + Simplifica la lógica compleja de transiciones.
  - + Provee una mejor visualización de alto nivel de los posibles estados y transiciones.
  - – Complica el diseño cuando la lógica de transición de estados ya es fácil de seguir.

Move Embellishment to Decorator



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).

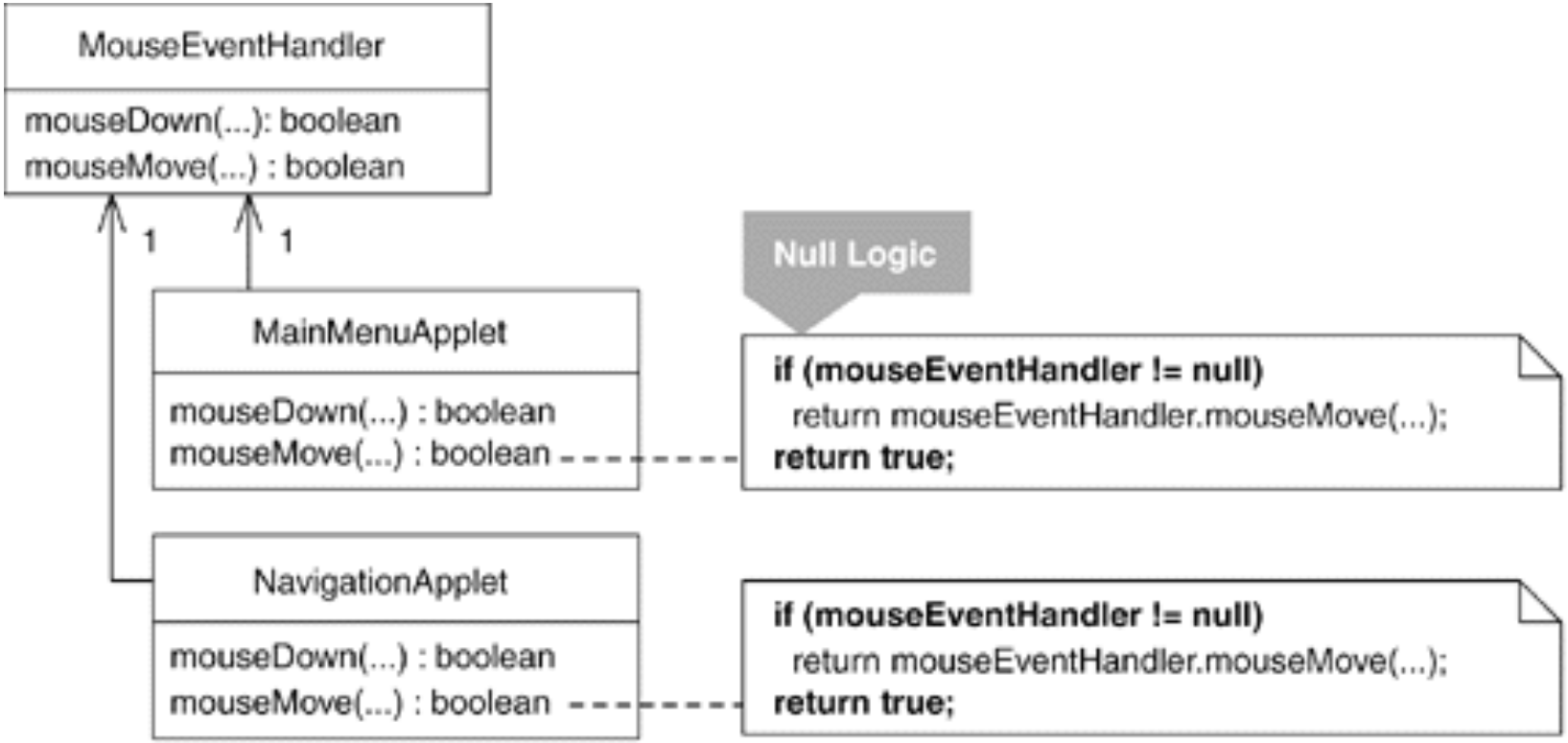


Alcanza? Si no sigo

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

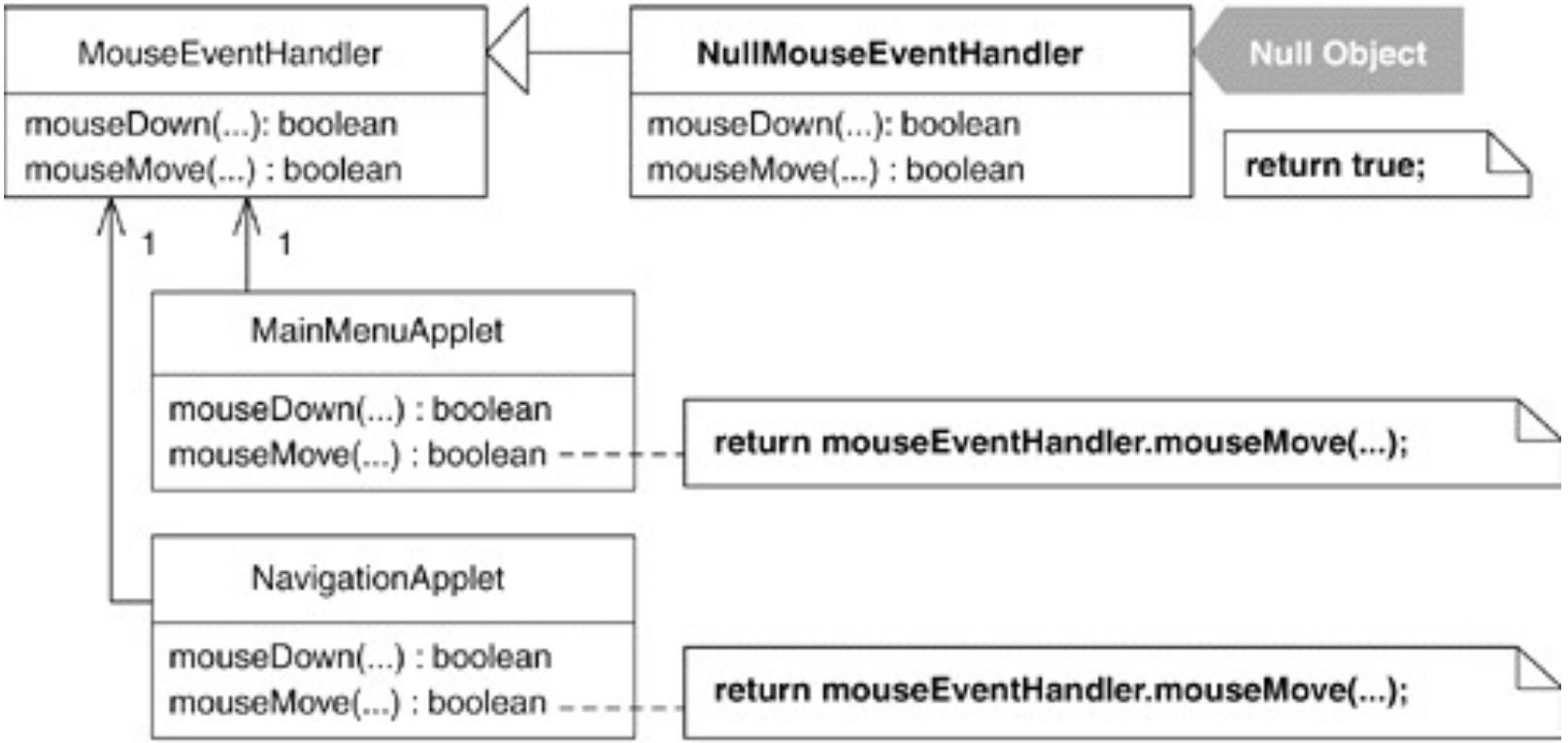
Introduce Null Object

- La lógica para manejarse con un valor nulo en una variable está duplicado por todo el código



Introduce Null Object

- Reemplazar la lógica de testeo por null con un Null Object



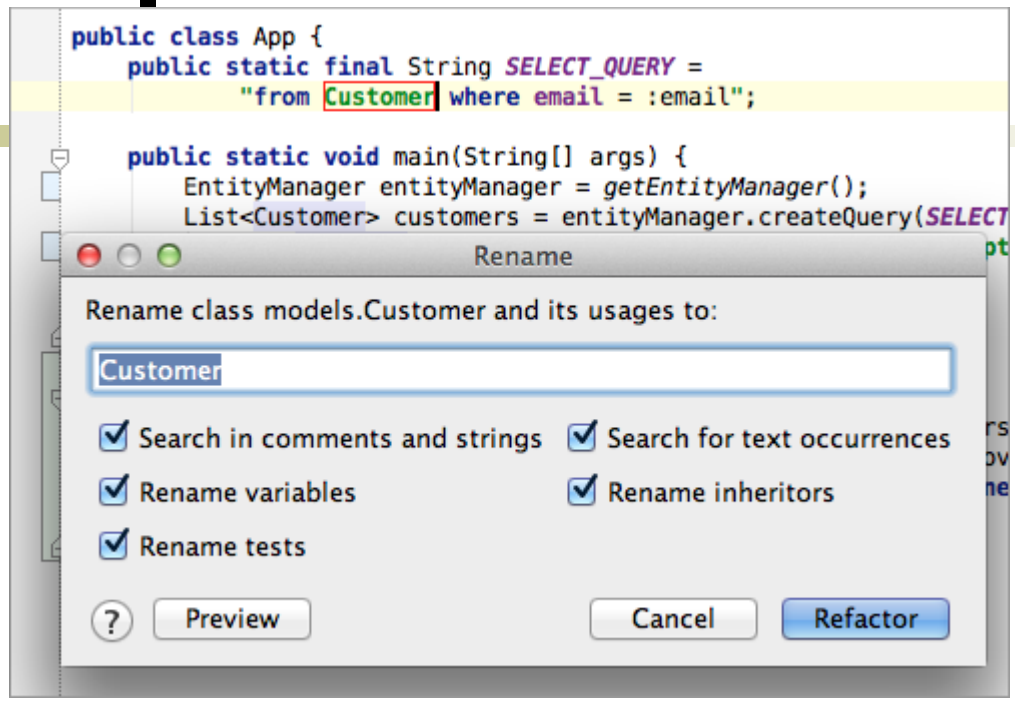
Automatización del refactoring

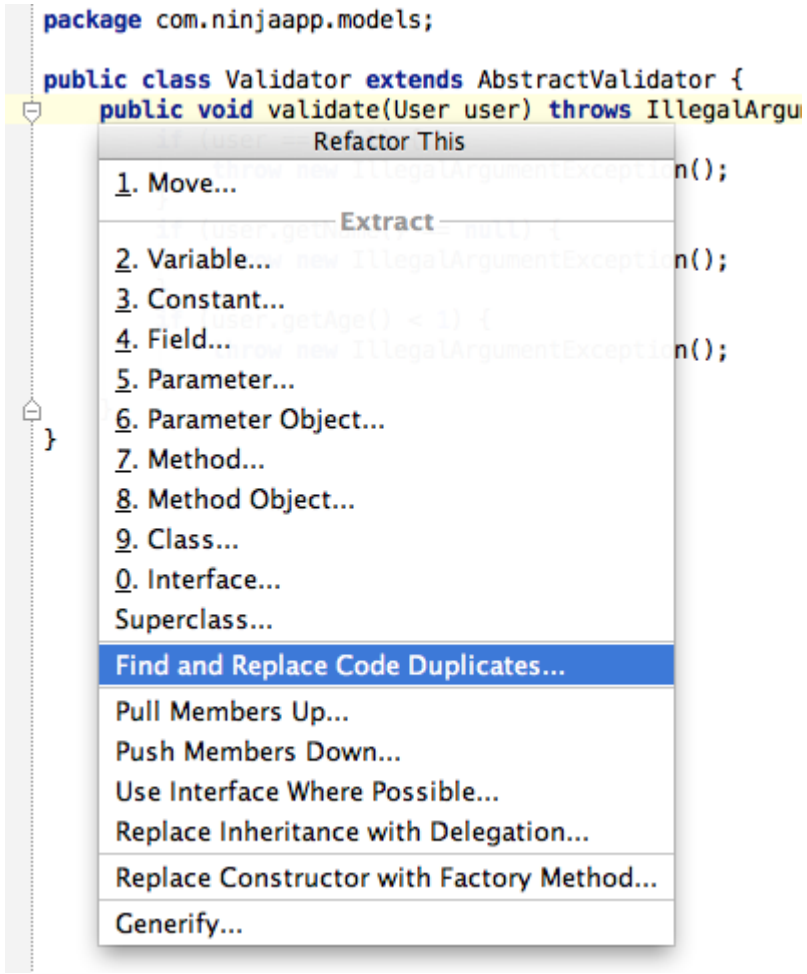
- Refactorizar a mano es demasiado costoso: lleva tiempo y puede introducir errores
- Herramientas de refactoring
- Características de las herramientas:
  - potentes para realizar refactorings útiles
  - restrictivas para preservar comportamiento del programa (uso de *precondiciones*)
  - interactivas, de manera que el chequeo de precondiciones no debe ser extenso

Las herramientas…

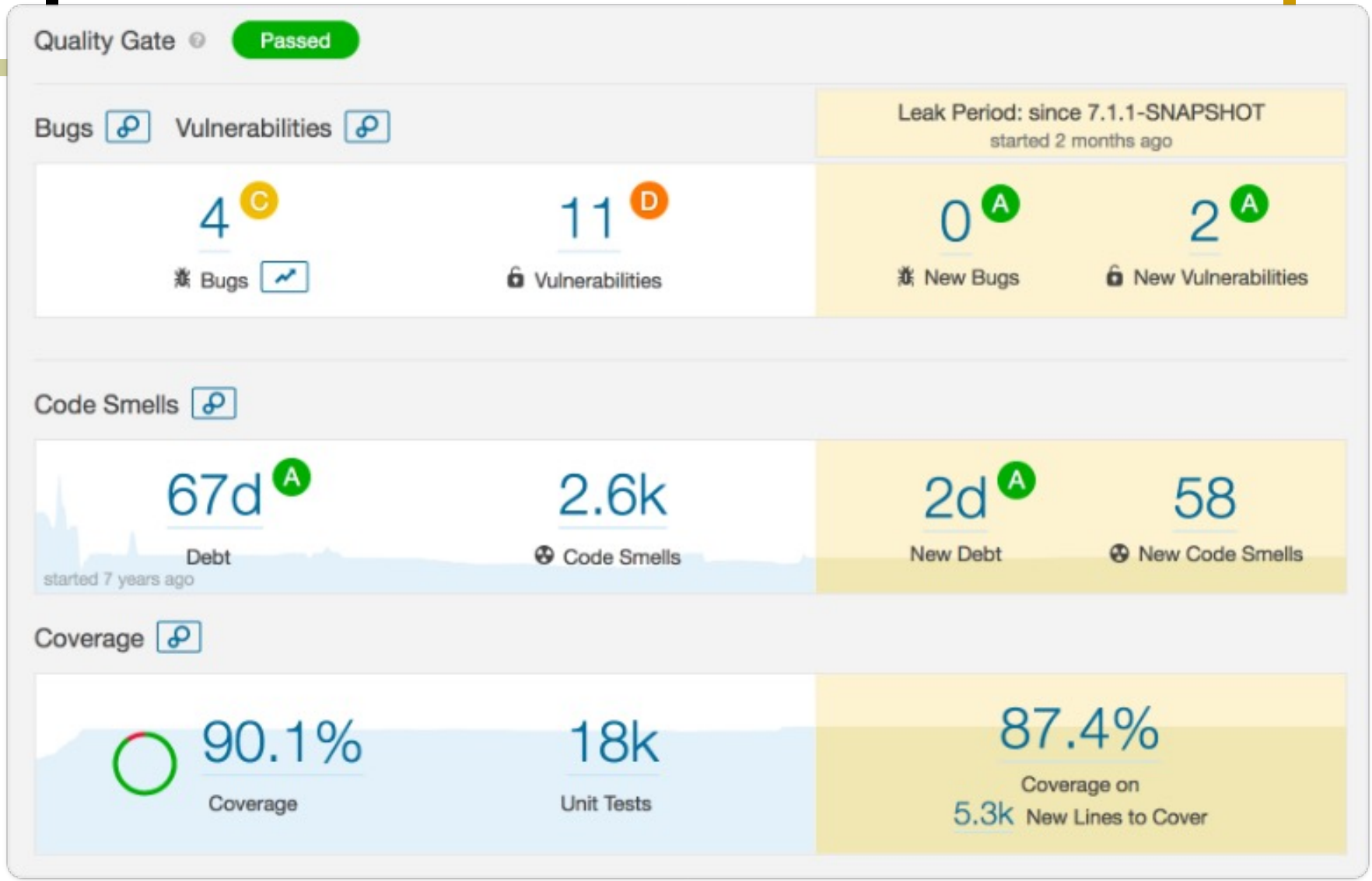
- Solo chequean lo que sea posible desde el árbol de sintaxis y la tabla de símbolos
- Pueden ser demasiado **conservativas** (no realizan un refactoring si no pueden asegurar preservación de comportamiento) o **asumir** buenas técnicas de programación
- Cada vez más potentes en la detección de code smells

IntelliJ IDEA





Sonar qube


Quality Gate
Passed
Bugs
Vulnerabilities
Leak Period: since 7.1.1-SNAPSHOT
started 2 months ago
4
C
Bugs
11
D
Vulnerabilities
0
A
New Bugs
2
A
New Vulnerabilities
Code Smells
67d
A
Debt
started 7 years ago
2.6k
Code Smells
2d
A
New Debt
58
New Code Smells
Coverage
90.1%
Coverage
18k
Unit Tests
87.4%
Coverage on
5.3k New Lines to Cover

Concepto asociado al refactoring: Deuda Técnica

- Concepto que introdujo Ward Cunningham para explicar a los managers la necesidad de refactoring
- Permite visualizar las consecuencias de un diseño “quick & dirty”
- **Capital** de la deuda: costo de remediar los problemas de diseño (costo del refactoring)
- **Interés** de la deuda: costo adicional en el futuro por mantener software con deuda técnica acumulada

Refactoring y testing

- Por un lado, el testing sirve para detectar si un refactoring cambió el comportamiento

Por otro lado, el refactoring puede romper un test aunque sea correcto

(Tesis de Maestría de Carlos Fontela: "Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras": http://sedici.unlp.edu.ar/handle/10915/29096

- Ejemplo 1: el refactoring cambió la API del SUT (nombre o signature del método)
- Ejemplo 2: el refactoring cambió las clases que se instancian en el test fixture

Ejemplo 1: cambia la API del SUT

- Refactoring Rename Method:
  - aunque lo haya aplicado automáticamente, las herramientas asumen un mundo cerrado
  - Tesis de grado de Juan Cruz Gardey: "Refactorings portables para soportar la evolución automática de código que utiliza componentes externos" http://sedici.unlp.edu.ar/handle/10915/72129
- Cualquier refactoring que cambie los parámetros de un método

Ejemplo 2: cambiaron las clases que se instancian en un test fixture

- Por ejemplo cuando aplicamos “Replace Conditional with Polymorphism”
  - Solución a- Definir un **Creation Method** en la clase Test Case (patrón de fixture setup del libro ”xUnit Test Patterns”): encapsular la complejidad de creación del fixture en un método con nombre apropiado que retorne instancias listas para usar
  - Solución b- Definir un *creation method* en la superclase de la jerarquía creada que reciba como parámetro un indicador de la subclase a instanciar (gralmente. int o string)

Aclaración

- Fowler en su libro llama “Replace Constructor with Factory Method” a la solucion b- , pero es un mal nombre porque no se debe confundir con el patrón Factory Method !

```
public class Jugador {
 static Jugador crear(String unNombre, String zona) {
   switch (zona) {
          case "A": return new JugadorZonaA();
          case "B": return new JugadorZonaB();
          case "C": return new JugadorZonaC();
   }
```

o

```
 static Jugador crear(String unNombre, String zona) {
   try {
      return (Jugador) Class.forName("JugadorZona" + zona).getConstructor().
                                                              newInstance();
   } catch (Throwable e) {
 ....
```