

Refactoring to patterns

Mejoras de diseño complejas
Qué? Cuándo? Cómo?

[Patrones de diseño]

- Qué son? Definición de GOF
- Cuáles vimos en clase? Clasificación
- Hay miles de patrones en diferentes áreas:
 - patrones de análisis (ej. Observations and Measurements)
 - comunicaciones (ej. Connector, Publisher-Subscriber)
 - concurrencia, distribución, web (conexión, presentación)
 - diseño de interfaces web (ej. Breadcrumbs, Carousel)
 - arquitectura de soft. (ej. MVC, Layers, Pipes & Filters)
 - sistemas distribuidos (ej. Broker, Bodyguard)
- Cómo usamos los patrones?

[Cuándo usamos patrones?]

- Consideraciones económicas
- Kent Beck: “Economic imperatives”:
 - Survival (supervivencia)
 - Net Present Value (ganar antes, gastar después)
 - Optionality (agregar valor es agregar opciones)
- Joshua Kerievsky: “over-engineering is as dangerous as under-engineering”
 - Over-engineering: construir software más sofisticado de lo que realmente necesita ser.
 - Under-engineering: construir software con un diseño pobre

[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]

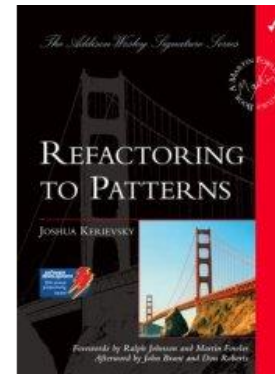
- Los patrones son tentadores para no quedarnos envueltos y arrastrar un mal diseño.
- También nos pueden llevar al otro extremo. Por esto es muy importante conocer las consecuencias tanto positivas como negativas de un patrón.

[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.



[Refactoring to patterns]

- El refactoring nos permite introducir patrones recién cuando el software que construimos evoluciona al punto que son necesarios
- No necesitamos adivinar o prever de antemano



Relación entre patrones y refactorings

“Patterns are where you want to be;
refactorings are ways to get there
from somewhere else.”
(Fowler, 2000)



[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

[Ejemplo del club de tenis]

- Imprimir los puntajes de cada set de un jugador en cada partido de tenis de una fecha específica.

Puntajes para los partidos de la fecha 7/5/2023

Partido:

Puntaje del jugador: Federico Delbonis: 6; 5; 7; Puntos del partido: 36

Puntaje del jugador: Guido Pella: 4; 7; 6; Puntos del partido: 34

Partido:

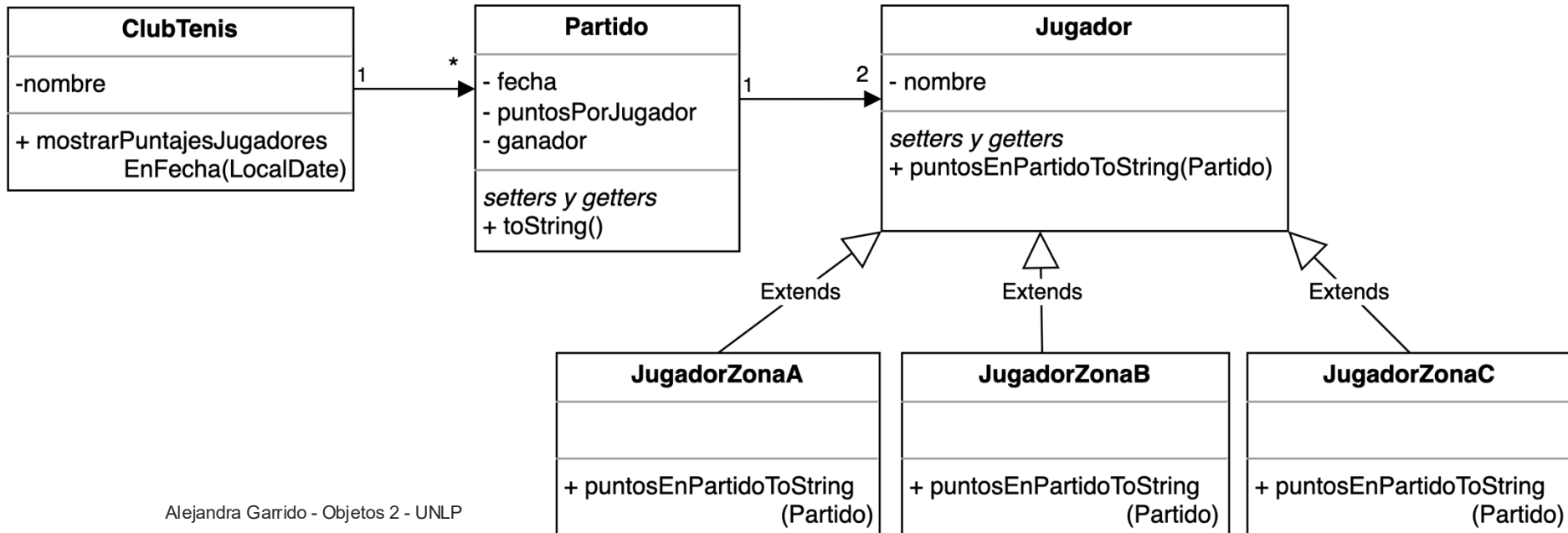
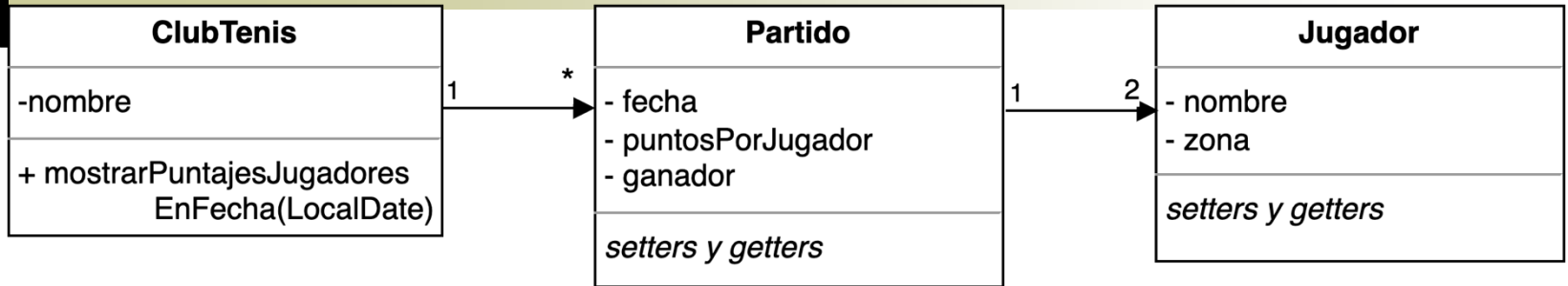
.....

```

public class ClubTennis {
    private String nombre;
    private List<Partido> coleccionPartidos;
    public String mostrarPuntajesJugadoresEnFecha(LocalDate fecha) {
        int totalGames = 0;
        List<Partido> partidosFecha;
        String result = "Puntajes para los partidos de la fecha " + fecha.toString() + "\n";
        partidosFecha = coleccionPartidos.stream().filter(p -> p.fecha().equals(fecha)).collect(Collectors.toList());
        for (Partido p : partidosFecha) {
            totalGames = 0;
            Jugador j1 = p.jugador1();
            result += "Partido: " + "\n";
            result += "Puntaje del jugador: " + j1.nombre() + ": ";
            for (int gamesGanados : p.puntosPorSetDe(j1)) {
                result += Integer.toString(gamesGanados) + ";";
                totalGames += gamesGanados;
            }
            result += "Puntos del partido: ";
            if (j1.zona() == "A") result += Integer.toString(totalGames * 2);
            if (j1.zona() == "B") result += Integer.toString(totalGames);
            if (j1.zona() == "C")
                if (p.ganador() == j1)
                    result += Integer.toString(totalGames);
                else
                    result += Integer.toString(0);

            Jugador j2 = p.jugador2();
            totalGames = 0;
            result += "Puntaje del jugador: " + j2.nombre() + ": ";
            for (int gamesGanados : p.puntosPorSetDe(j2)) {
                result += Integer.toString(gamesGanados) + ";";
                totalGames += gamesGanados;
            }
            result += "Puntos del partido: ";
            if (j2.zona() == "A") result += Integer.toString(totalGames * 2);
            if (j2.zona() == "B") result += Integer.toString(totalGames);
            if (j2.zona() == "C")
                if (p.ganador() == j2)
                    result += Integer.toString(totalGames);
                else
                    result += Integer.toString(0);
        }
        return result;
    }
}

```



[Secuencia de refactorings que aplicamos en el camino]

- Extract Method
- Move Method x 2
- Replace Conditional with Polymorphism
- Replace Temp with Query

Supongamos que llegamos a esta situación

```
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



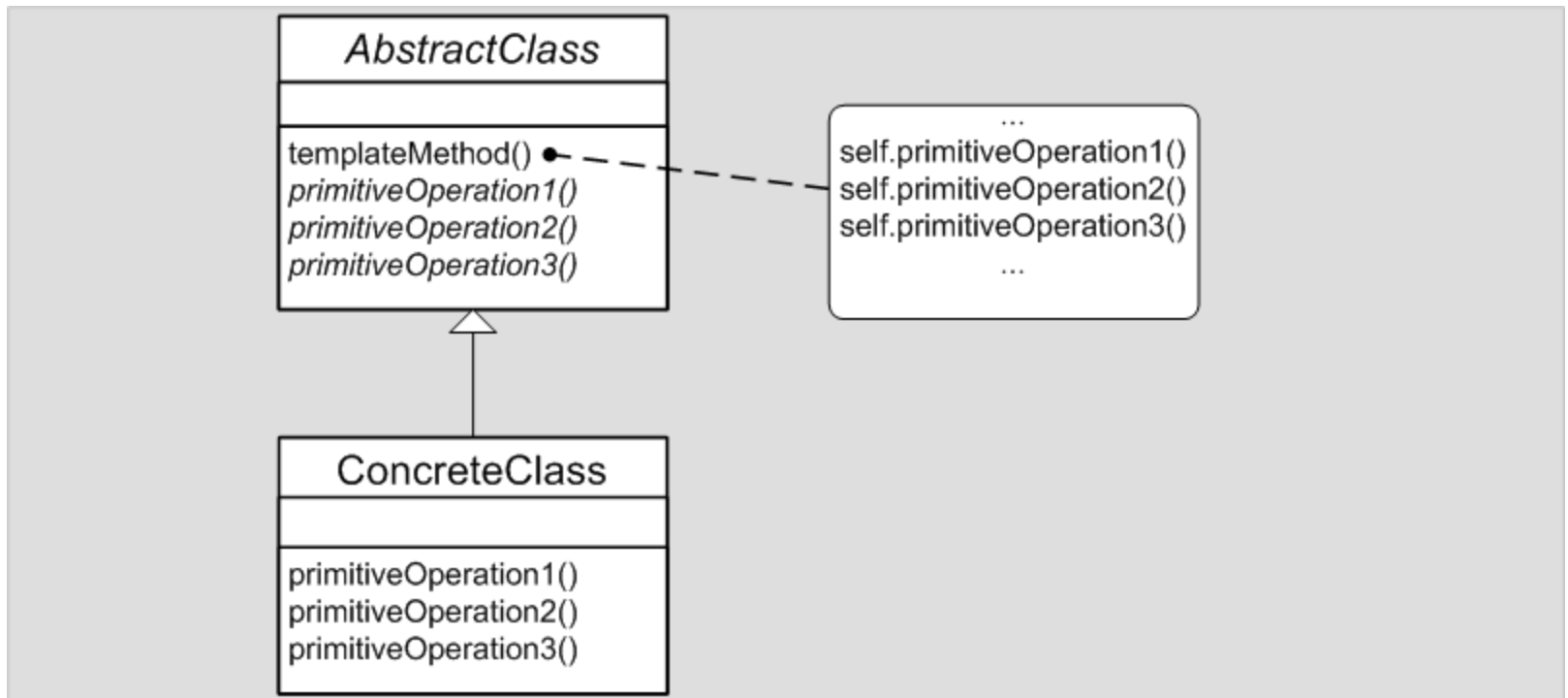
[Soluciones a Código Duplicado]

- Extract Method + Pull Up Method

[Qué otra posibilidad?]

- Hay pasos en el algoritmo de mostrar los puntajes que se repiten en todas las subclases
- ¿Qué otro patrón solucionaría esta situación de código duplicado?

[Patrón Template Method]

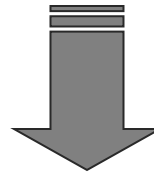


Propósito de un patrón vs. Problema que soluciona

- 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?

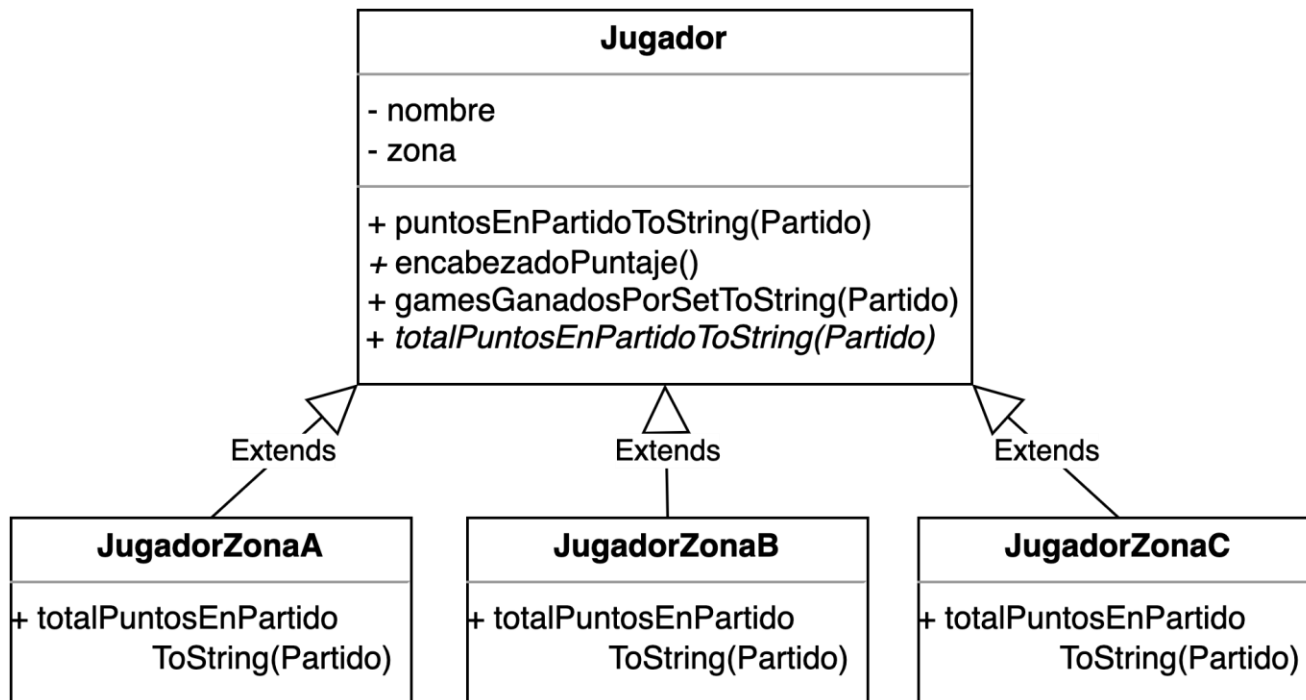


```
public class JugadorZonaA {  
    String puntosEnPartidoToString(Partido partido) {  
        String result = "Puntaje del jugador: " + nombre() + ": ";  
        for (int gamesGanados: partido.puntosPorSetDe(this))  
            result += Integer.toString(gamesGanados) + ";";  
        result +=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

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

[Volviendo atrás en el ejemplo]

```
Jugador>>puntosEnPartidoToString(Partido partido)
```

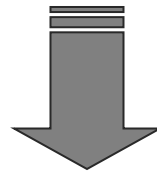
```
...
```

```
    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);
```

- Supongamos que antes de hacer subclases de Jugador hubiéramos sabido que un jugador puede cambiar de zona
- ¿Hay algún patrón que permite desacoplar un algoritmo en otro objeto y cambiar ese algoritmo dinámicamente?

[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.

[Primeros pasos]

```
public class ZonaJugadorStrategy { ... }
```

```
public class Jugador {  
    public Jugador(String unNombre) {  
        nombre = unNombre;  
        zona = new ZonaJugadorStrategy();  
    }  
    public 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(zona.puntosGanadosEnPartido(?));  
        return result;  
    }  
}
```



Ajustar antes de mover: Replace Temp with Query

```
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?
- Es una decisión importante a tomar, para no introducir nuevos code smells (por ej. Long Parameter List; Inappropriate Intimacy; etc.)



[3) Extract Parameter

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



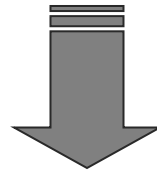
```
public class Jugador {  
    public Jugador(String unNombre, ZonaJugador unaZona) {  
        nombre = unNombre;  
        zona = unaZona;  
    }  
}
```


[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 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]

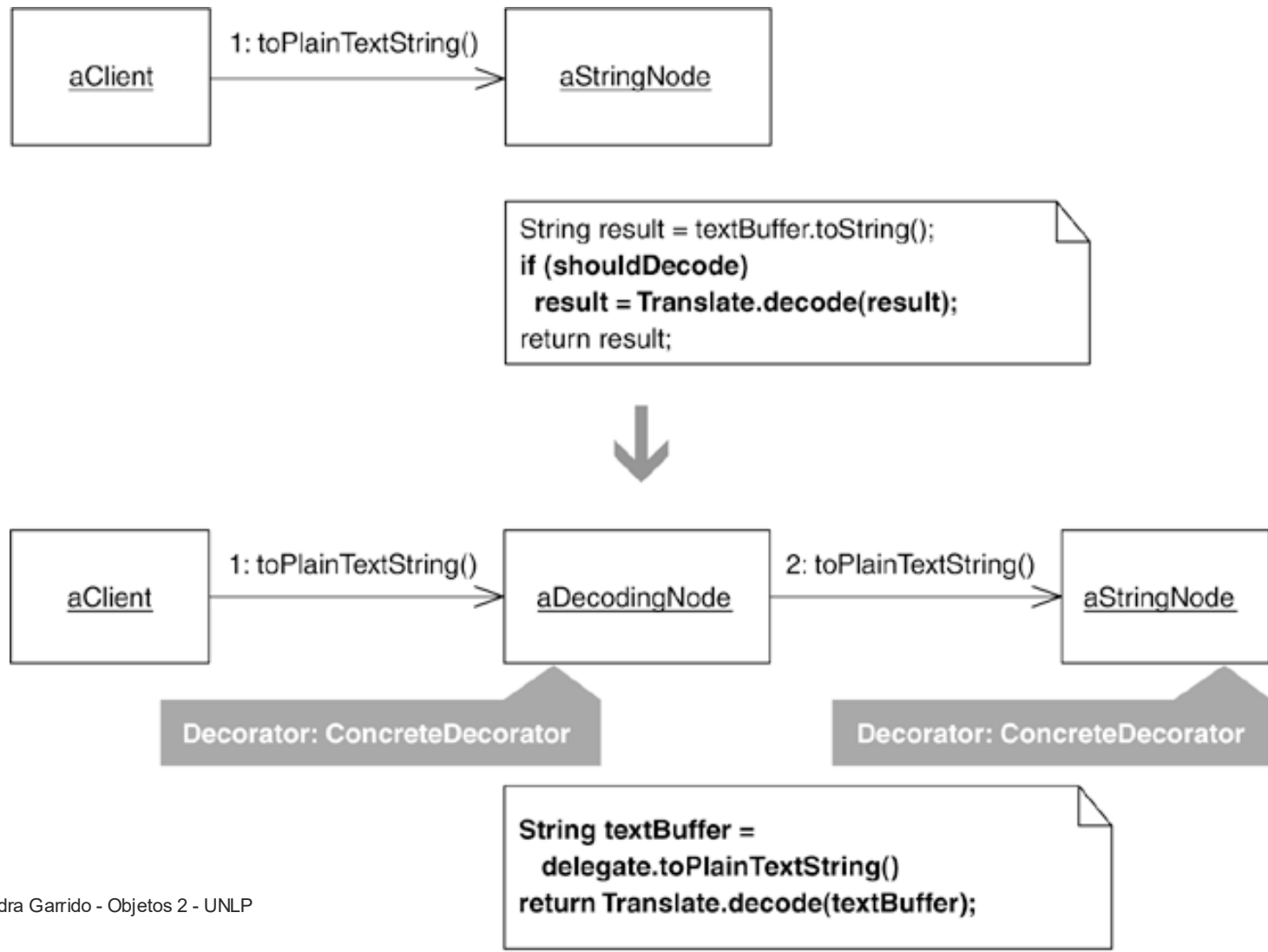
■ Motivación

- 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]

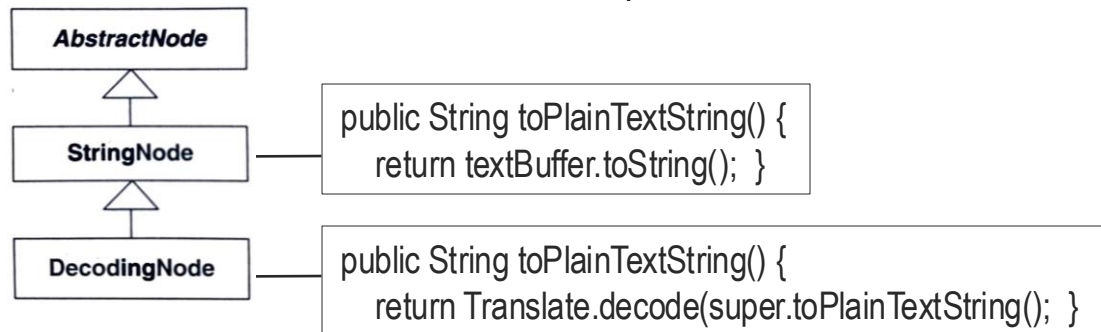
1. Aplicar “*Replace Type-Code with Class*” para crear una clase que será la superclase del State a partir de la v.i. que mantiene el estado
2. Aplicar “*Extract Subclass*” [F] para crear una subclase del State por cada uno de los estados de la clase contexto.
3. Por cada método de la clase contexto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

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

[Veamos un nuevo problema]

```
class Robot {  
    ...  
    boolean defend(Territory myTerritory) {  
        if (weapon != null)  
            weapon.move(myTerritory...)  
    ...}  
  
    boolean attack(Robot enemy) {  
        if (weapon != null)  
            weapon.aim(...  
    }  
    boolean upgrade() {  
        if (weapon != null)  
            ...  
    }
```

[Fuerzas del problema]

- Hay lógica condicional que testea por null en muchos métodos de mi clase
- La lógica condicional testea la existencia o no de un colaborador para delegarle una tarea
- Quisiéramos poder tratar al null como un objeto más, que simplemente no hace nada al recibir el mensaje
- De esta forma también evitamos el condicional y los errores o excepciones que se producen al olvidarnos de chequear por null

[Patrón Null Object]

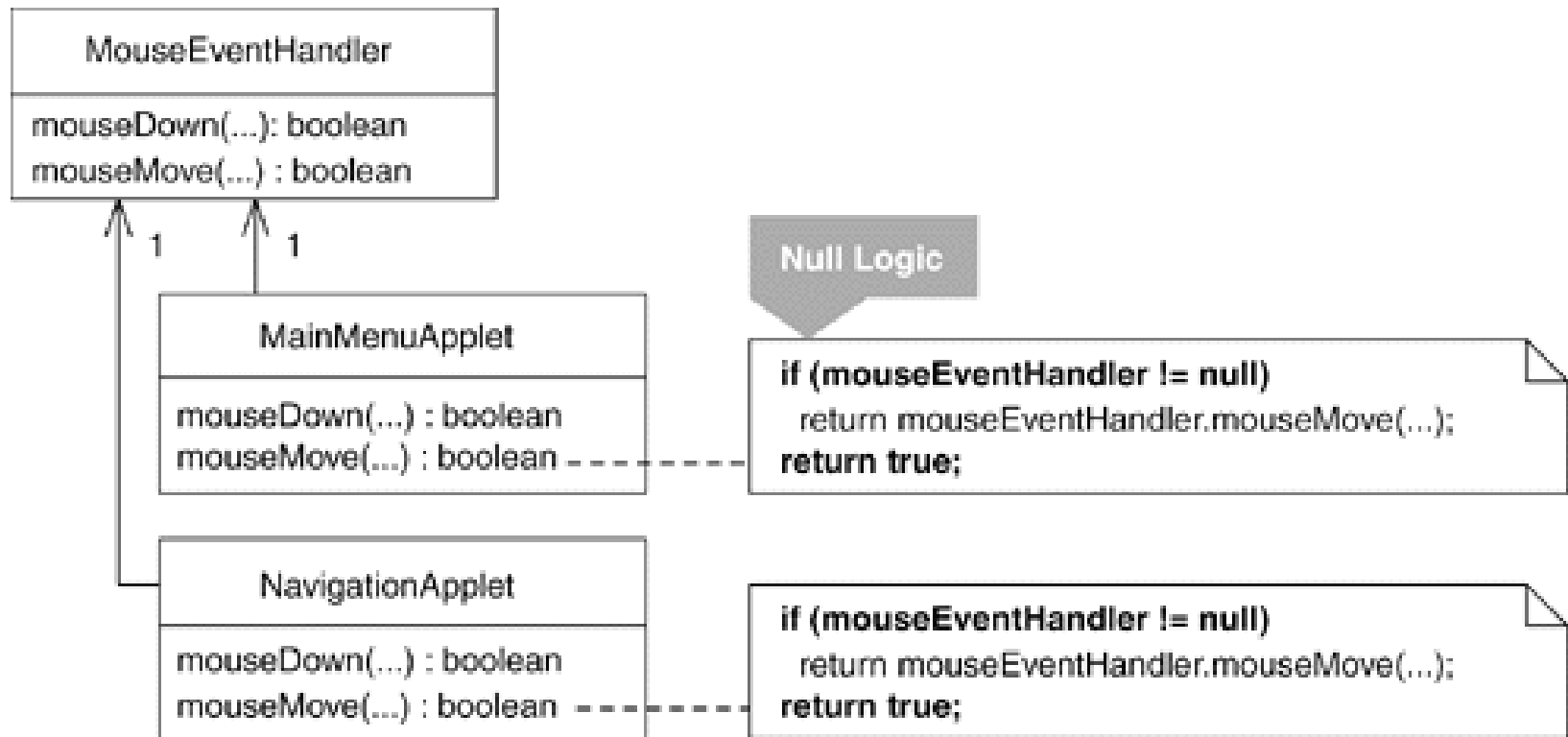
- **Propósito:**
proporcionar un sustituto para otro objeto que comparte la misma interfaz, pero no hace nada. El objeto nulo encapsula las decisiones de implementación sobre cómo "no hacer nada" y oculta esos detalles a sus colaboradores.
- **Aplicabilidad:**
 - un objeto tiene un colaborador, que algunas veces no hace nada
 - queremos que el objeto cliente pueda ignorar la diferencia del colaborador que hace algo con el que no hace nada
 - queremos reusar el comportamiento de hacer nada

[Consecuencias]

- Define jerarquías de clase que consisten de objetos reales y null objects. En cualquier lugar que el cliente espera un objeto real también puede tomar un null object
- Hace el código del cliente más simple. Los clientes pueden tratar a sus colaboradores de manera uniforme
- Encapsula el comportamiento de hacer nada en un objeto, que puede ser reusado por múltiples clientes
- Requiere crear una clase NullObject cada vez que en una jerarquía necesitemos el comportamiento nulo
- Podría ser difícil de implementar si la usan varios clientes y no hay un acuerdo de cual debería ser el comportamiento nulo

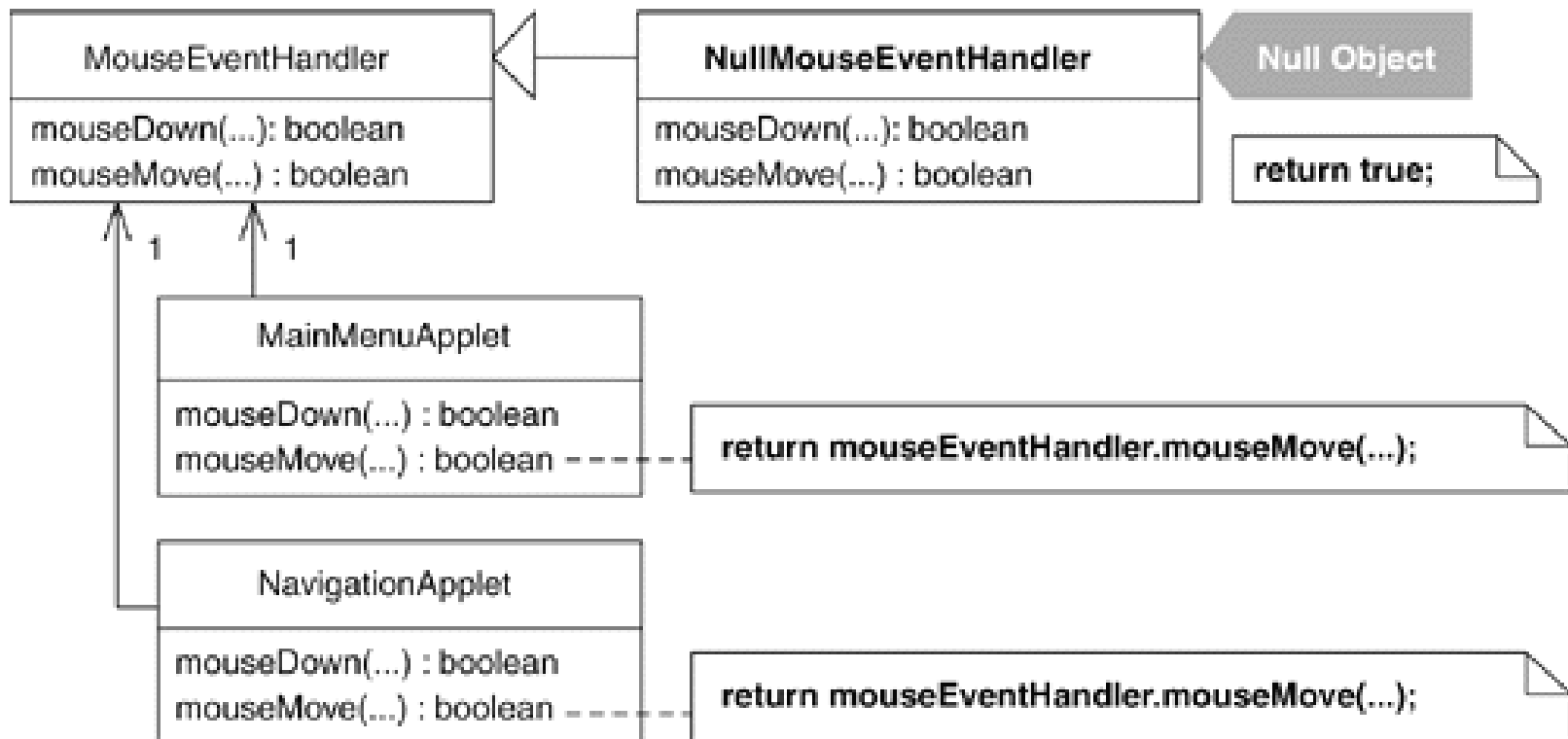
[Introduce Null Object]

- 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



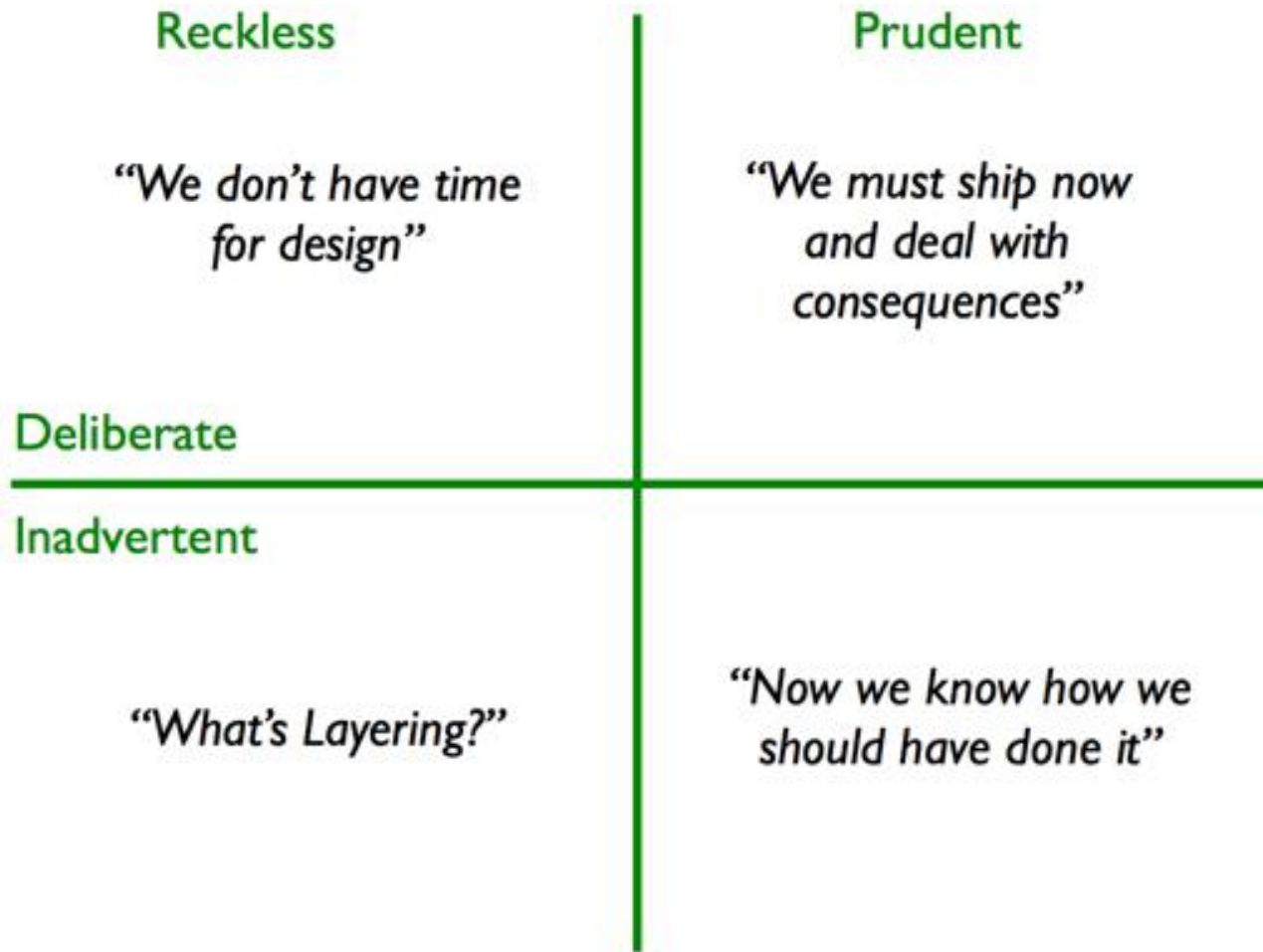
Proceso de diseño propuesto por los métodos ágiles



[Deuda Técnica (TD: Technical Debt)]

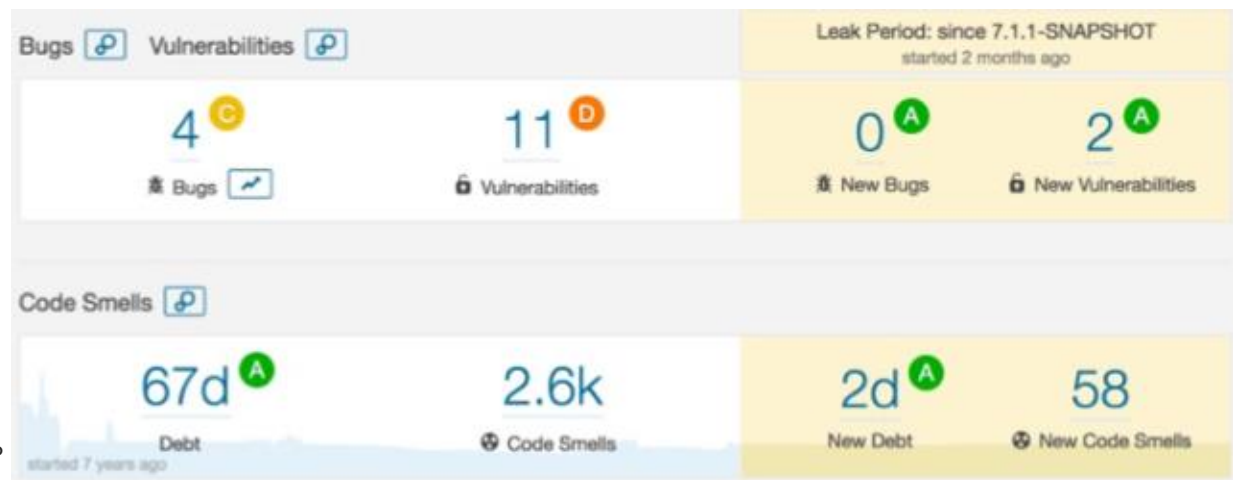
- Concepto que introdujo Ward Cunningham para explicar a los stakeholders la necesidad de refactoring
- Está bien tomar prestado o endeudarse cuando *estratégicamente* conviene entregar código rápido para ganar feedback y aprender, y el código refleja nuestro entendimiento actual del problema
- El peligro es cuando esa deuda no se paga!
- Cada minuto dedicado a código que acarrea deuda cuenta como interés. Y hasta que no se aplique refactoring para pagar esa deuda, seguiremos acumulando interés
- El código que escribimos debería ser lo suficientemente limpio para refactorizarlo fácilmente a medida que lo entendemos mejor

[TD Quadrant. Martin Fowler]



[TD. Conceptos asociados]

- **Capital** de la deuda: costo de remediar los problemas de diseño (costo del refactoring)
- **Interés** de la deuda: costo adicional o esfuerzo extra por acarrear un mal diseño
- Varias IDEs tienen la capacidad de cuantificar y visualizar el capital de la TD



[Referencias]

- Pattern-Oriented Software Architecture. A System of Patterns. Bushcmann et al. Wiley.
- UI patterns: <https://ui-patterns.com/patterns>
- Empirical Software Design: When & Why. Kent Beck. ACM Tech Talks. [Link](#)
- Refactoring to Patterns. Joshua Kerievsky. Addison-Wesley.
- Debt Metaphor. Ward Cunningham. [Video](#)