Programación Orientada a Objetos (POO)
FCEyN - UBA

Hernán Wilkinson

Twitter: @HernanWilkinson

If

- Debemos respetar sintaxis "*objeto mensaje*"
- If como "keyword" implica que el lenguaje no es de objetos
- If se implementa con polimorfismo en lenguajes de clasificación
- Usar If implica que no estamos usando polimorfismo →
  - Diseño menos mantenible
  - Diseño NO orientado a objetos

If – Cómo sacarlo

1. Crear una jerarquía polimorfica con una abstracción por cada "condición"
2. Usando el mismo nombre de mensaje repartir el "cuerpo del if" en cada abstracción (usar polimorfismo)
3. Nombrar el mensaje del paso anterior
4. Nombrar las abstracciones
5. Reemplazar "if" por envío de mensaje polimórfico
6. Buscar objeto polimórfico si es necesario

If – Cómo sacarlo

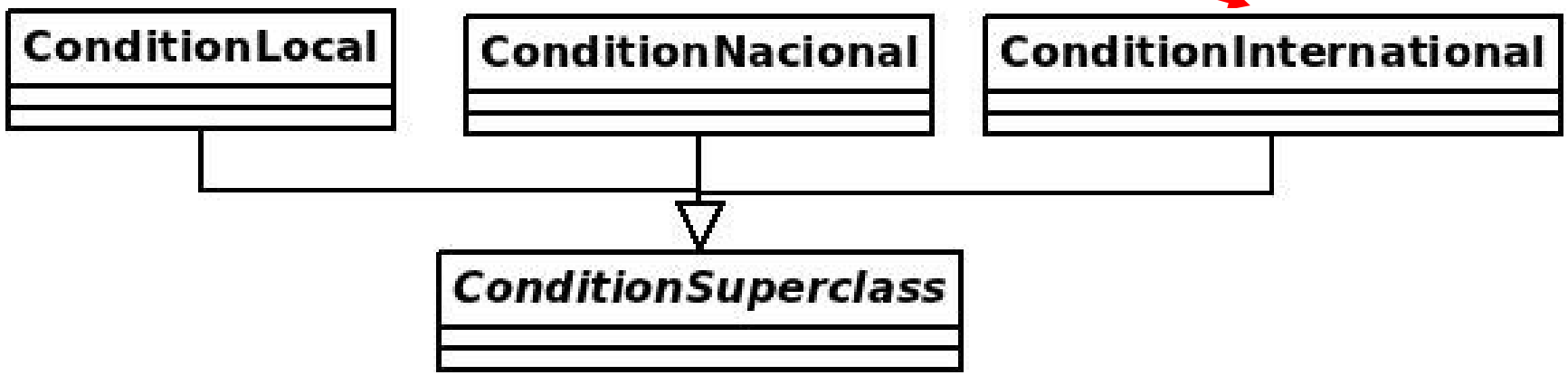
1. Crear una jerarquía polimórfica con una abstracción por cada "condición"

```
if (call.isLocal())
    calculateLocalCostOf(call);
else if (call.isNational())
    calculateNationalCostOf(call);
else if (call.isInternational())
    calculateInternationalCostOf(call);
```

If – Cómo sacarlo

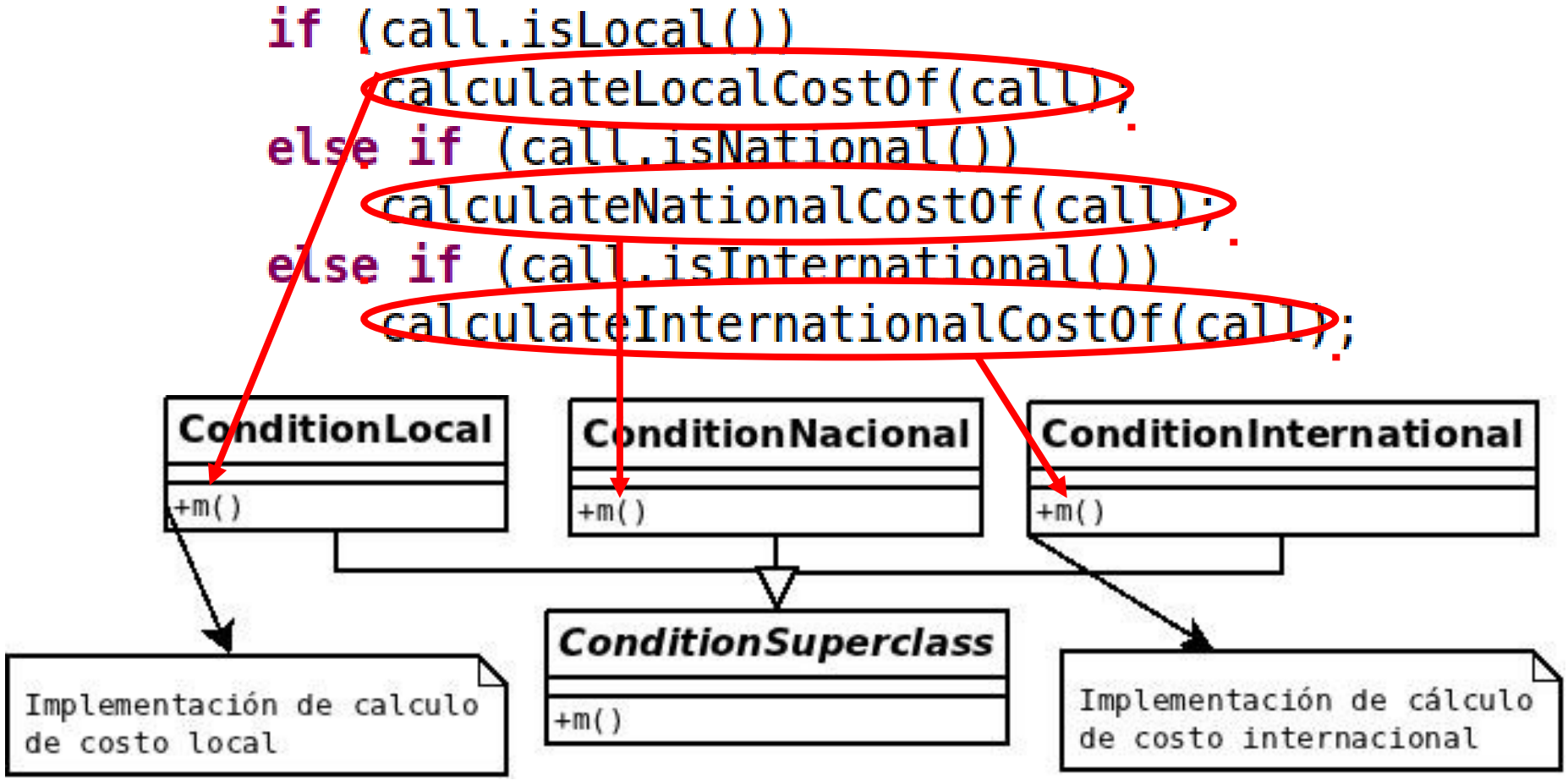
1. Crear una jerarquía polimórfica con una abstracción por cada "condición"

```
if (call.isLocal())
    calculateLocalCostOf(call);
else if (call.isNational())
    calculateNationalCostOf(call);
else if (call.isInternational())
    calculateInternationalCostOf(call);
```



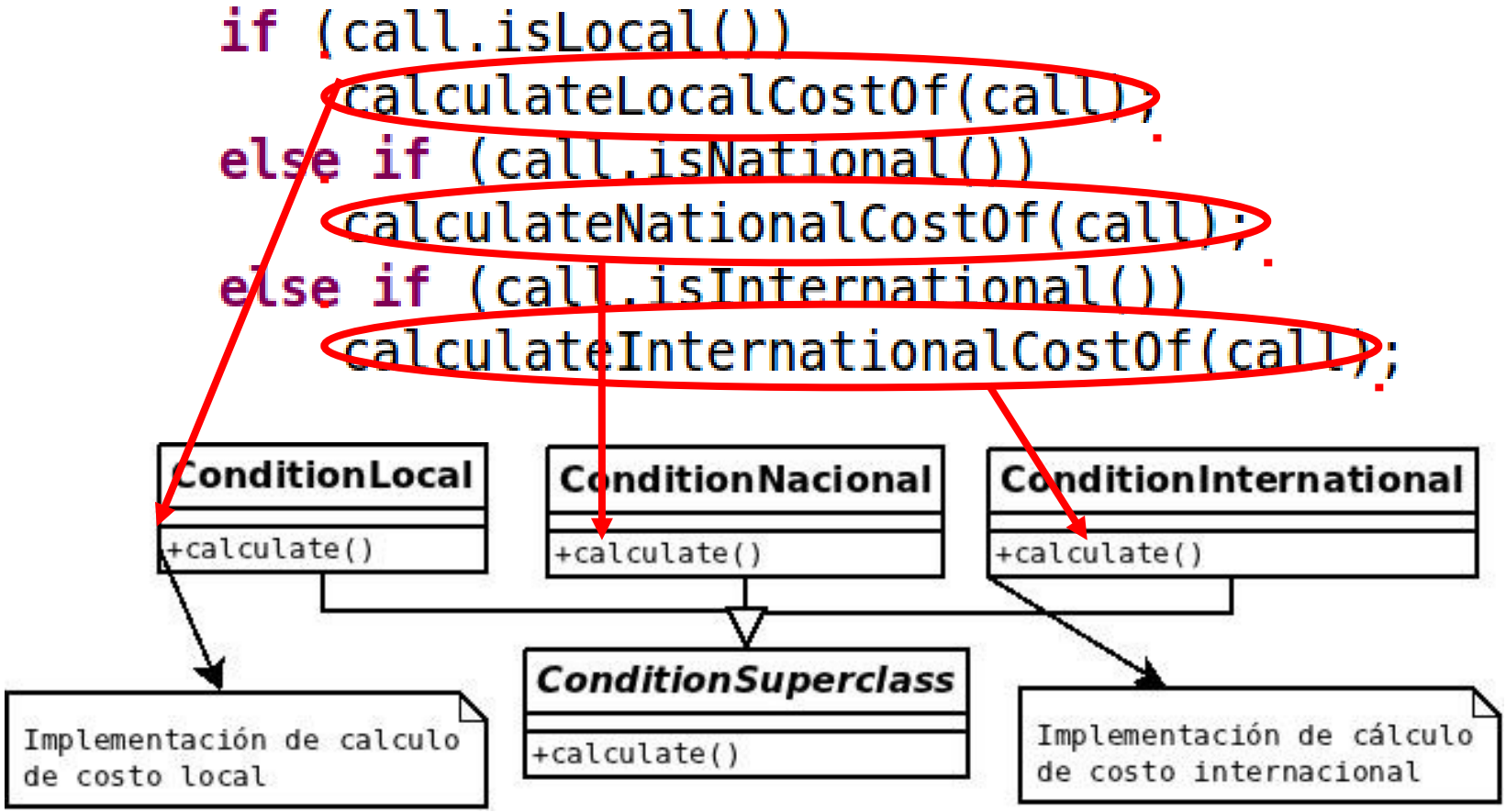
If – Cómo sacarlo

2. Usando el mismo nombre de mensaje repartir el "cuerpo del if" en cada abstracción



If – Cómo sacarlo

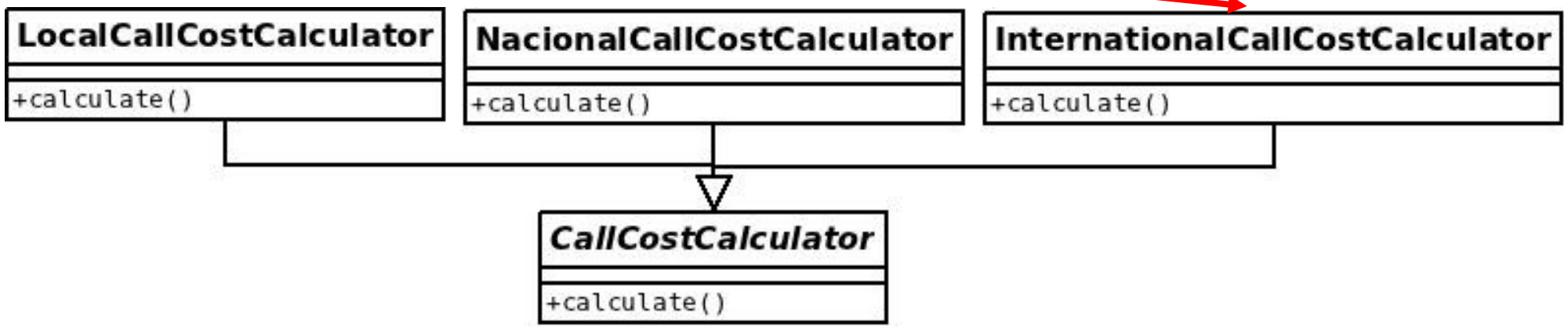
3. Nombrar el mensaje del paso anterior



If – Cómo sacarlo

4. Nombrar las abstracciones

```
if (call.isLocal())
    calculateLocalCostOf(call);
else if (call.isNational())
    calculateNationalCostOf(call);
else if (call.isInternational())
    calculateInternationalCostOf(call);
```

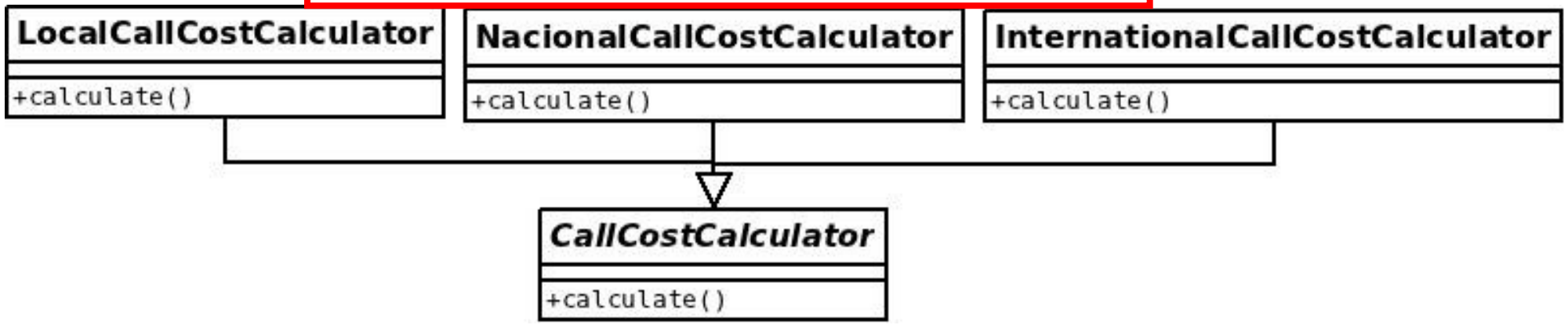


If – Cómo sacarlo

5. Reemplazar "if" por envío de mensaje polimórfico

```
if (call.isLocal())
    calculateLocalCostOf(call);
else if (call.isNational())
    calculateNationalCostOf(call);
else if (call.isInternational())
    calculateInternationalCostOf(call);
```

```
costCalculator.calculate();
```

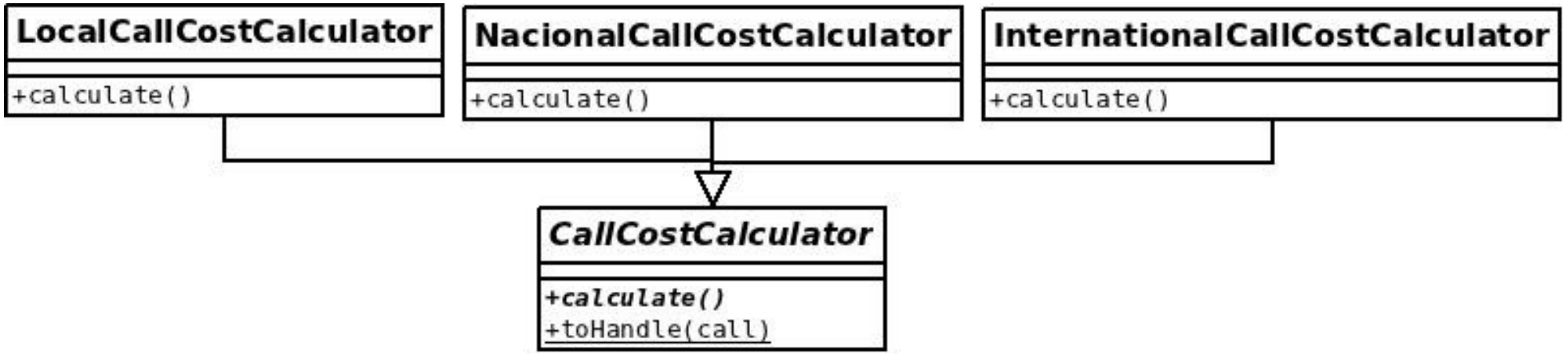


If – Cómo sacarlo

6. Buscar objeto polimórfico si es necesario

```
CostCalculator costCalculator = CostCalculator.toHandle(call);

costCalculator.calculate();
```



If - Conclusiones

- Los objetos toman la "decisión"
- La decisión no está hardcodeada
- "Sumun" del diseño
  - Aparece algo nuevo en el dominio de problema, aparece algo nuevo en mi modelo
- Puedo hacer meta-programación por ejemplo para saber cuantos "costeadores" hay
  - CostCalculator allSubclasses size
- **Límite**:
  - No se puede sacar cuando colaboran objetos de distinto dominio
  - Ej.: if (account.balance()=0) …
- **Ver Regla 10**

Principios Básicos de Diseño

- Simplicidad
  - KISS, The Hollywood Principle,Single Responsibility Principle
- Consistencia
  - Modelo mental, Metáfora
- Entendible
  - Legibilidad, Mapeo con dominio de Problema
- Máxima Cohesión
  - Objetos bien funcionales (SRP)
- Mínimo Acoplamiento
  - Minimizar “ripple effect”
- … y otros más…

Heurísticas de Diseño

- **H1**: Cada ente del dominio de problema debe estar representado por un objeto

- Las ideas son representadas con una sola clase (a menos que se soporte la evolución de ideas)
- Los entes pueden tener una o más representaciones en objetos, depende de la implementación
- La esencia del ente es modelado por los mensajes que el objeto sabe responder

Heurísticas de Diseño

- **H2**: Los objetos deben ser cohesivos representando responsabilidades de un solo dominio de problema

- Cuanto más cohesivo es un objeto más reutilizable es

Heurísticas de Diseño

- **H3**: Se deben utilizar buenos nombres, que sinteticen correctamente el conocimiento contenido por aquello que están nombrando

- Los nombres son el resultado de sintetizar el conocimiento que se tiene de aquello que se está nombrando
- Los nombres que se usan crean el vocabulario que se utiliza en el lenguaje del modelo que se está creando

Heurísticas de Diseño

- **H4**: Las clases deben representar conceptos del dominio de problema

- Las clases no son módulos ni componentes de reuso de código
- Crear una clase por cada "componente" de conocimiento o información del dominio de problema
- La ausencia de clases implica ausencia de conocimiento y por lo tanto la imposibilidad del sistema de referirse a dicho conocimiento

Heurísticas de Diseño

- **H5**: Se deben utilizar clases abstractas para representar conceptos abstractos

- Nunca denominar a las clases abstractas con la palabra *Abstract*. Ningún concepto se llama "Abstract…"

Heurísticas de Diseño

- **H6**: La clases no-hojas del árbol de subclasificación deben ser clases abstractas

- Evitar definir métodos de tipo *final* o no *virtual* en clases abstractas puesto que impiden la evolución del modelo

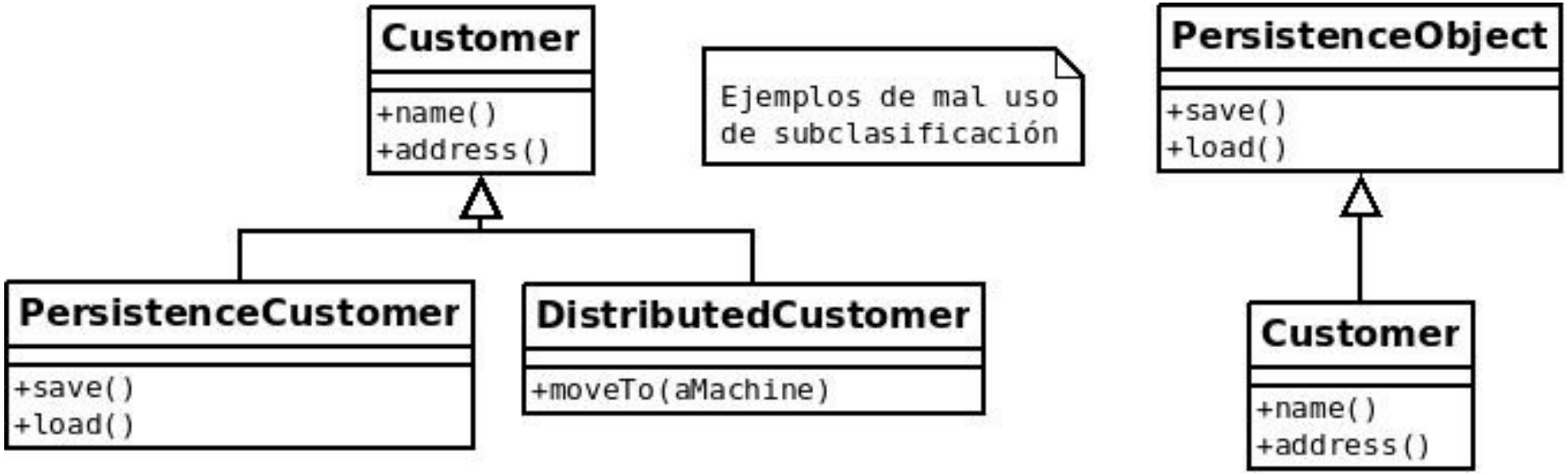
Heurísticas de Diseño

- **H7**: Evitar definir variables de instancia en las clases abstractas porque esto impone una implementación en todas las subclases

- Definir variables de instancia de tipo *private* implica encapsulamiento a nivel "módulo" y no a nivel objeto. Encapsulamiento a nivel objeto implica variables de instancia tipo *protected*

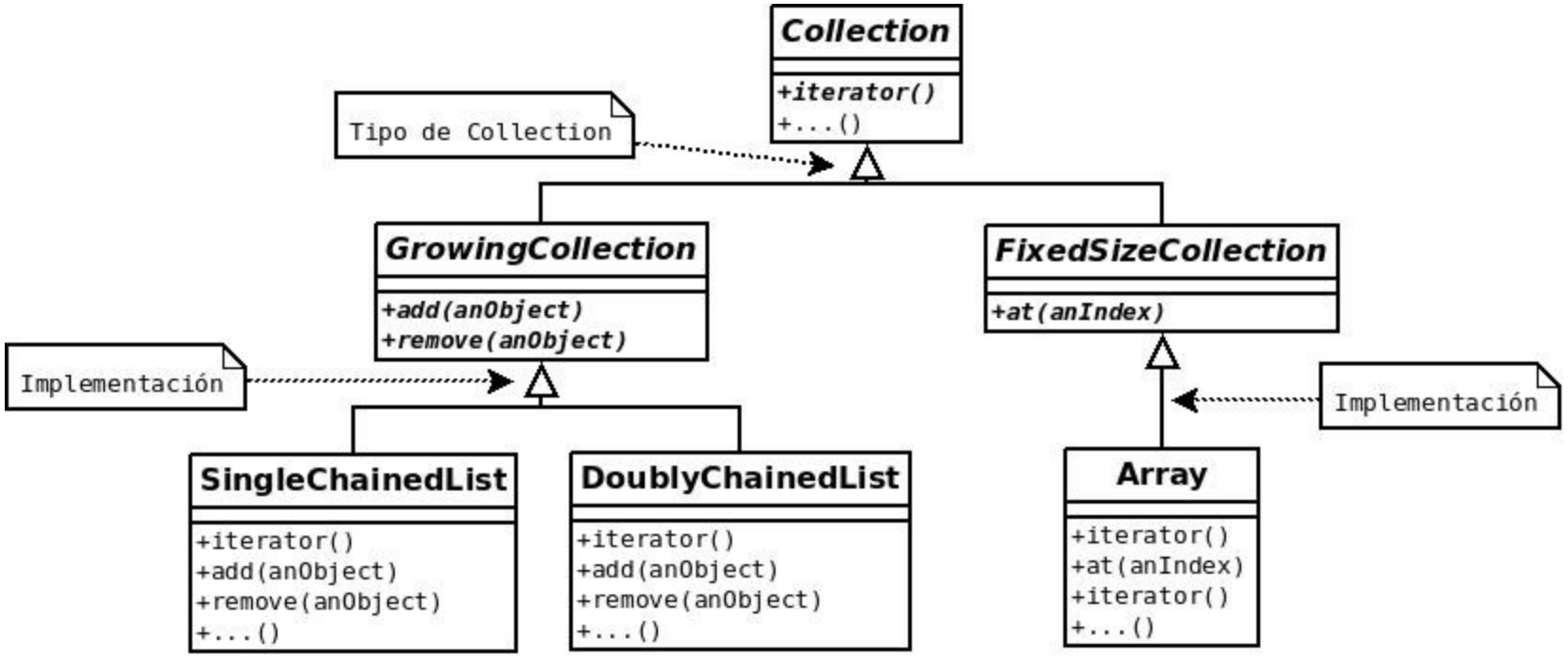
Heurísticas de Diseño

- **H8**: El motivo de subclasificación debe pertenecer al dominio de problema que se esta modelando



Heurísticas de Diseño

- **H9**: No se deben mezclar motivos de subclasificación al subclasificar una clase



Heurísticas de Diseño

- **H10**: Reemplazar el uso de if con polimorfismo.

- El *if* en el paradigma de objetos es implementado usando polimorfismo
- Cada *if* es un indicio de la falta de un objeto y del uso del polimorfismo

Heurísticas de Diseño

H11: Código repetido refleja la falta de algún objeto que represente el motivo de dicho código

- Código repetido no significa "texto repetido"
- Código repetido significa patrones de colaboraciones repetidas
- Reificar ese código repetido y darle una significado por medio de un nombre

Heurísticas de Diseño

- **H12**: Un Objeto debe estar completo desde el momento de su creación

- El no hacerlo abre la puerta a errores por estar incompleto, habrá mensajes que no sabe resonder
- Si un objeto está completo desde su creación, siempre responderá los mensajes que definió

Heurísticas de Diseño

- **H13**: Un Objeto debe ser válido desde el momento de su creación

- Un objeto debe representar correctamente el ente desde su inicio
- Junto a la regla anterior mantienen el modelo consistente constantemente

Heurísticas de Diseño

- **H14**: No utilizar *nil* (o *null*)

- *nil* (o *null*) no es polimórfico con ningún objeto
- Por no ser polimórfico implica la necesidad de poner un *if* lo que abre la puerta a errores
- *nil* es un objeto con muchos significados por lo tanto poco cohesivo
- Las dos reglas anteriores permiten evitar usar *nil*

Heurísticas de Diseño

- **H15**: Favorecer el uso de objetos inmutables

- Un objeto debe ser inmutable si el ente que representa es inmutable
- La mayoría de los entes son inmutables
- Todo modelo mutable pude ser representado por uno inmutable donde se modele los cambios de los objetos por medio de eventos temporales

Heurísticas de Diseño

- **H16**: Evitar el uso de setters

- Para aquellos objetos mutables, evitar el uso de setters porque estos puede generar objetos inválidos
- Utilizar un único mensaje de modificación como *syncronizeWith(anObject)*

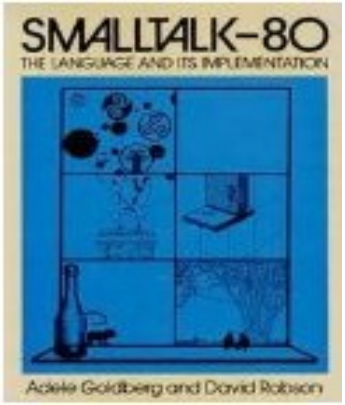
Heurísticas de Diseño

- **H17**: Modelar la arquitectura del sistema

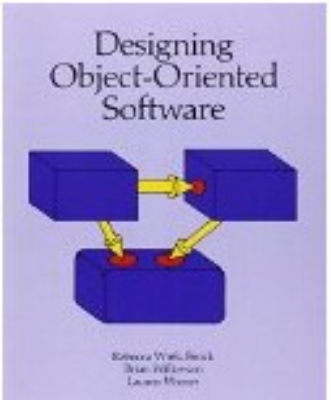
- Crear un modelo de la arquitectura del sistema (subsistemas, etc)
- Otorgar a los subsistemas la responsabilidad de mantener la validez de todo el sistema (la relación entre los objetos)
- Otorgar la responsabilidad a los subsistemas de modificar un objeto por su impacto en el conjunto

Bibliografía y referencias

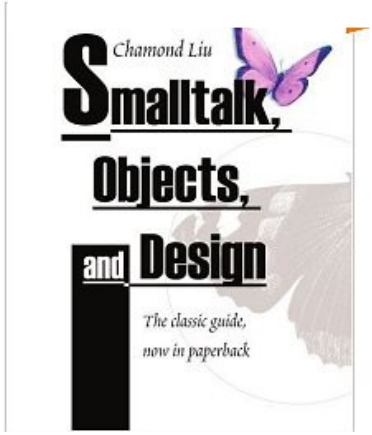
Bibliografía recomendada



Smalltalk-80: The language and its Implementation
A. Goldberg et al



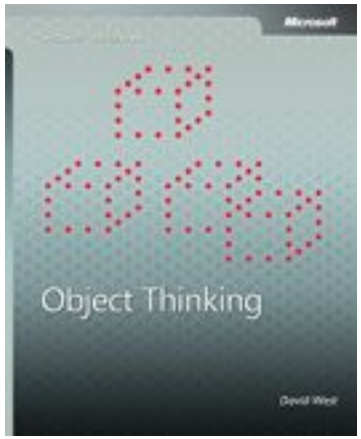
Designing Object-Oriented Software
Rebecca Wirfs-Brook



Smalltalk, Objects and Design
Chamond Lie

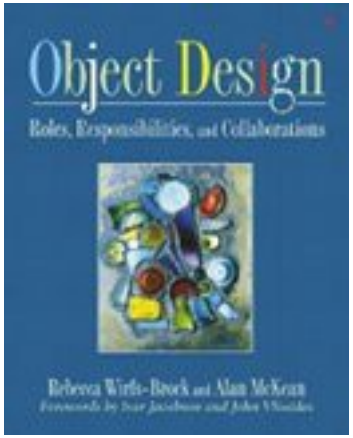
Smalltalk Best Practice Patterns.
Kent Beck

Bibliografía recomendada



Object Thinking.
David West, 2004.

Object-Oriented Software Construction.
Bertrand Meyer, 2000.



Object Design: Roles, Responsibilities, and Collaborations.
R.Wirfs-Brock, A.McKean, 2002.