Ingeniería de Software I
Diseño de Software con Objetos
Parte II
UBA - FCEyN

Hernán Wilkinson

Twitter: @HernanWilkinson

Paradigma de Objetos

- **Polimorfismo**: Dos o más objetos son polimórficos entre si para un conjunto de mensajes, si responden a dicho conjunto de mensajes semánticamente igual

- Semánticamente igual significa:
  - Hacen lo mismo
  - Reciben objetos polimórficos
  - Devuelven objetos polimórficos

Paradigma de Objetos

- **Prototipo**
  - Objeto ejemplar que representa el comportamiento de un conjunto de objetos similares
  - Se utiliza como mecanismo de representación de conocimiento en leguajes de prototipación o "*Wittgestein-nianos*"

Paradigma de Objetos

- **Clase**
  - Objeto que representa un concepto o idea del dominio de problema
  - Por ser un objeto, sabe responder mensajes
  - Existe como mecanismo de representación de conocimiento en leguajes de clasificación o Aristotélicos
  - No existe en lenguajes de prototipación

Paradigma de Objetos

- **Subclasificación**
  - Herramienta utilizada para organizar el conocimiento en ontologías
  - Es una relación “estática” entre clases
  - Permite organizar el conocimiento y representarlo en clases abstractas (que representan conceptos abstractos) y clases concretas (que representan conceptos con realizaciones concretas)

Paradigma de Objetos

- **Pseudo Variable “super”**
  - super = self → true
    o sea, referencian el mismo objeto
  - super indica al method lookup que la búsqueda debe empezar a partir de la superclase de la clase en la cual está definido el método

Paradigma de Objetos

- Problemas de la clasificacion
  - Relación “estática” (entre clases)
  - Obliga a tener una clase y por lo tanto su nombre antes del objeto concreto, lo cual es antinatural
  - Obliga a “generalizar” cuando aún no se posee el conocimiento total de aquello que representa

Paradigma de Objetos

- Problemas de la subclasificación
  - Debe ser especificada de manera inversa a como se obtiene el conocimiento
  - Rompe el encapsulamiento puesto que la subclase debe conocer la implementación de la superclase

Paradigma de Objetos

- **Tipo**: Conjunto de mensajes

- No está más relacionado con temas de "espacio" en memoria (ej. int, long, etc)
- No es únicamente un nombre, sino que define semántica
- Hay lenguajes que lo reifican como Java en la construcción "*interface*"

Paradigma de Objetos

- Cuándo se realiza
  - Estáticamente vs. Dinámicamente
- Qué se hace al romperse la validación:
  - Fuerte vs. Débil

Cuándo/Qué	Estáticamente (Compilación)	Dinámicamente (Ejecución)
Fuerta (Strong)	Java, C#, Pascal	Smalltalk, Ruby
Débil (Weak)	C, C++	VB6

Paradigma de Objetos

- **Tipar una Variable**:
  - Asignarle a dicha variable el tipo de los objetos que estará referenciando
  - Como es “tipo”, debe utilizarse la construcción del lenguaje para tal fin (ej. *interface*)
- **Clasear una Variable**:
  - Asignarle a dicha variable no sólo el tipo sino también una posición en la jerarquía de clases que los objetos que estará referenciando serán instancia de
  - Mayor acoplamiento que “tipar una variable”
  - Define no sólo tipo sino también implementación

Paradigma de Objetos

- En lenguajes “estáticos” siempre se debe “tipar” y nunca “clasear”
- Problema de las interfaces:
  - Aparecen al final del desarrollo, cuando todas las variables están “claseadas” y no existe refactoring para hacer el cambio
  - Crear una *interface* por cada clase (absurdo y tedioso)
  - Impone grandes limitaciones al momento de extender los modelos

Paradigma de Objetos

- En lenguajes “dinámicos” el único acoplamiento entre objetos son los mensajes que se envían
- Cada mensaje es un “tipo”
- Al haber menos acoplamiento → mayor facilidad de cambiar el modelo

Bloques (Closure)

- Lambda Expression:
  - Proviene del Lambda Calculus
  - Se utiliza para definir “Funciones sin nombre”
  - En objetos: Conjunto de colaboraciones
- Closure:
  - Lambda expression bindeada a un contexto particular (ej, el contexto en el cual es “instanciada”)
  - Ejemplo: Un método es un closure en el contexto del objeto receptor (porque “self” o “this” está bindeado al objeto receptor del mensaje)

Closure - Smalltalk

```
10 factorial -> 3628800.
```

Devuelve el factorial de 10

```
[ 10 factorial ] -> [10 factorial].
```

Devuelve un objeto que representa que se le enviará el mensaje factorial a 10

```
[ 10 factorial ] value -> 3628800.
```

Se evalua el “closure” y por lo tanto se obtiene el factorial de 10

Closure - Smalltalk

```
| aClosure |

aClosure := [10 factorial ].

aClosure value -> 3628800.
```

Como es un objeto, se lo puede asignar a una variable…

… y se le pueden enviar mensajes

```
| aClosure |

aClosure := [ :anInteger | anInteger factorial ].

aClosure value: 10 -> 3628800.
```

Puede recibir “colaboradores”

… y por lo tanto pasárselos

Closure - Smalltalk

```
exampleClosure4

    | var1 |

    var1 := 1.

    ^ [ :anInteger | var1 := anInteger * var1 ].
```

Se bindea al contexto en el cual fue creado

```
exampleClosure5

    | aClosure |

    aClosure := self exampleClosure4.

    aClosure value: 5. "Retorna 5"
    aClosure value: 5 "Retorna 25"
```

Por lo tanto lo modifica cuando se lo evalúa

Closure - Conclusiones

- ¿Qué son?
  - Objetos
- ¿Qué representan?
  - Conjunto de colaboraciones (como los métodos!)
- ¿Para qué se usan?
  - ¡Para parametrizar “comportamiento” (colaboraciones, código)
  - Para referenciar al contexto donde fueron creados

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

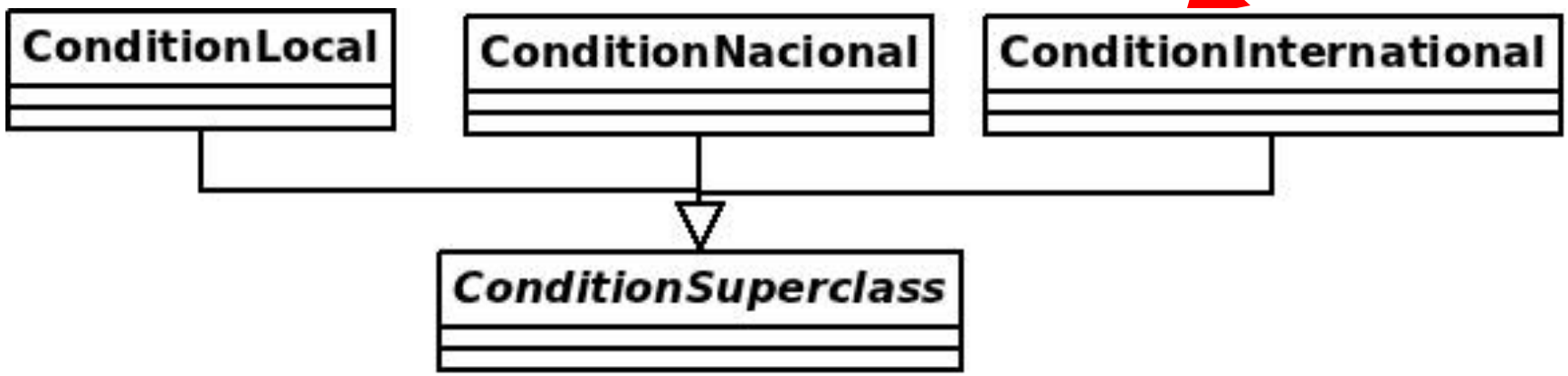
1. Crear una jerarquía polimórfica con una abstracción por cada “condición”

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

If – Cómo sacarlo

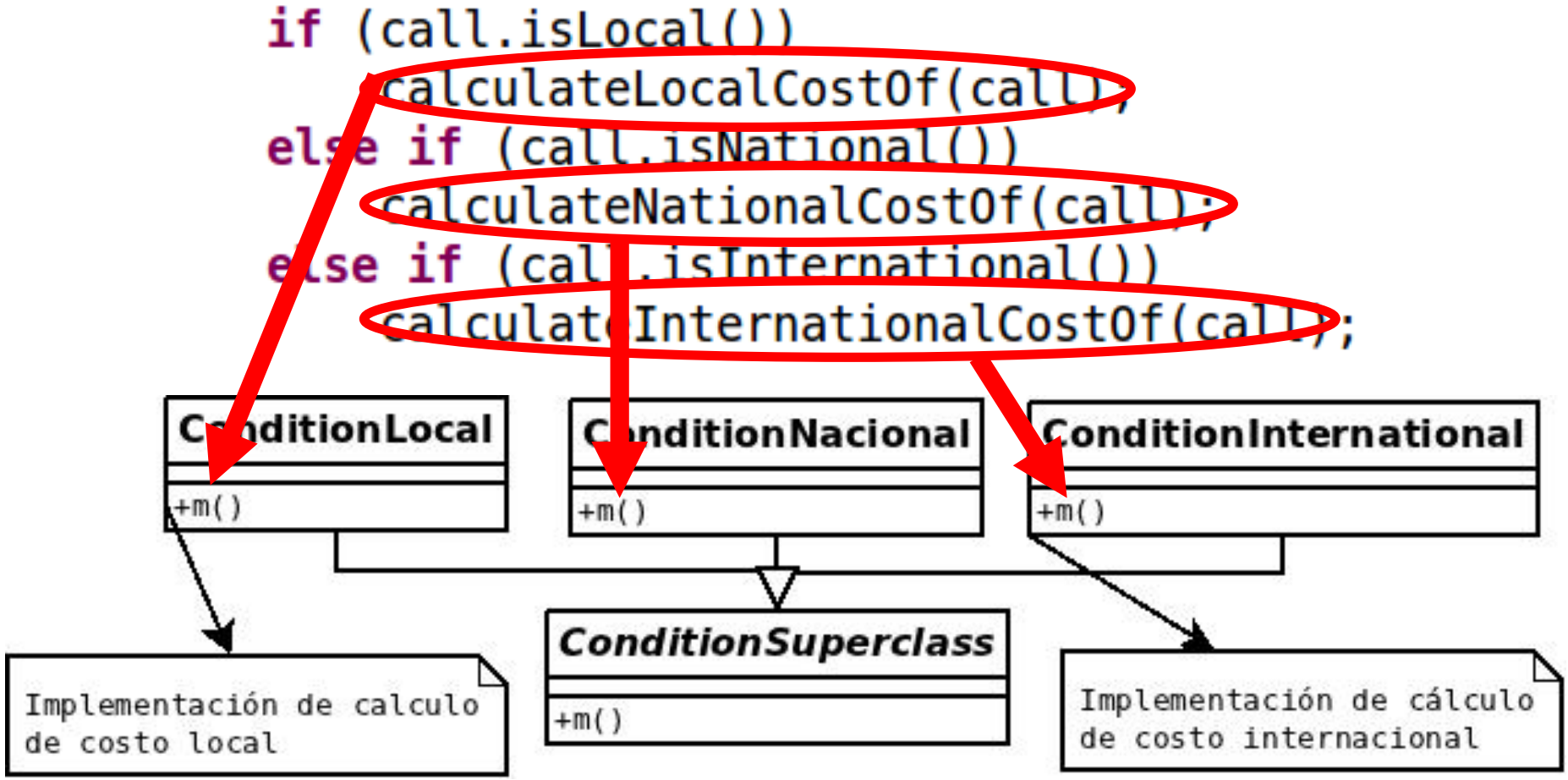
1. Crear una jerarquía polimórfica con una abstracción por cada "condición"

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



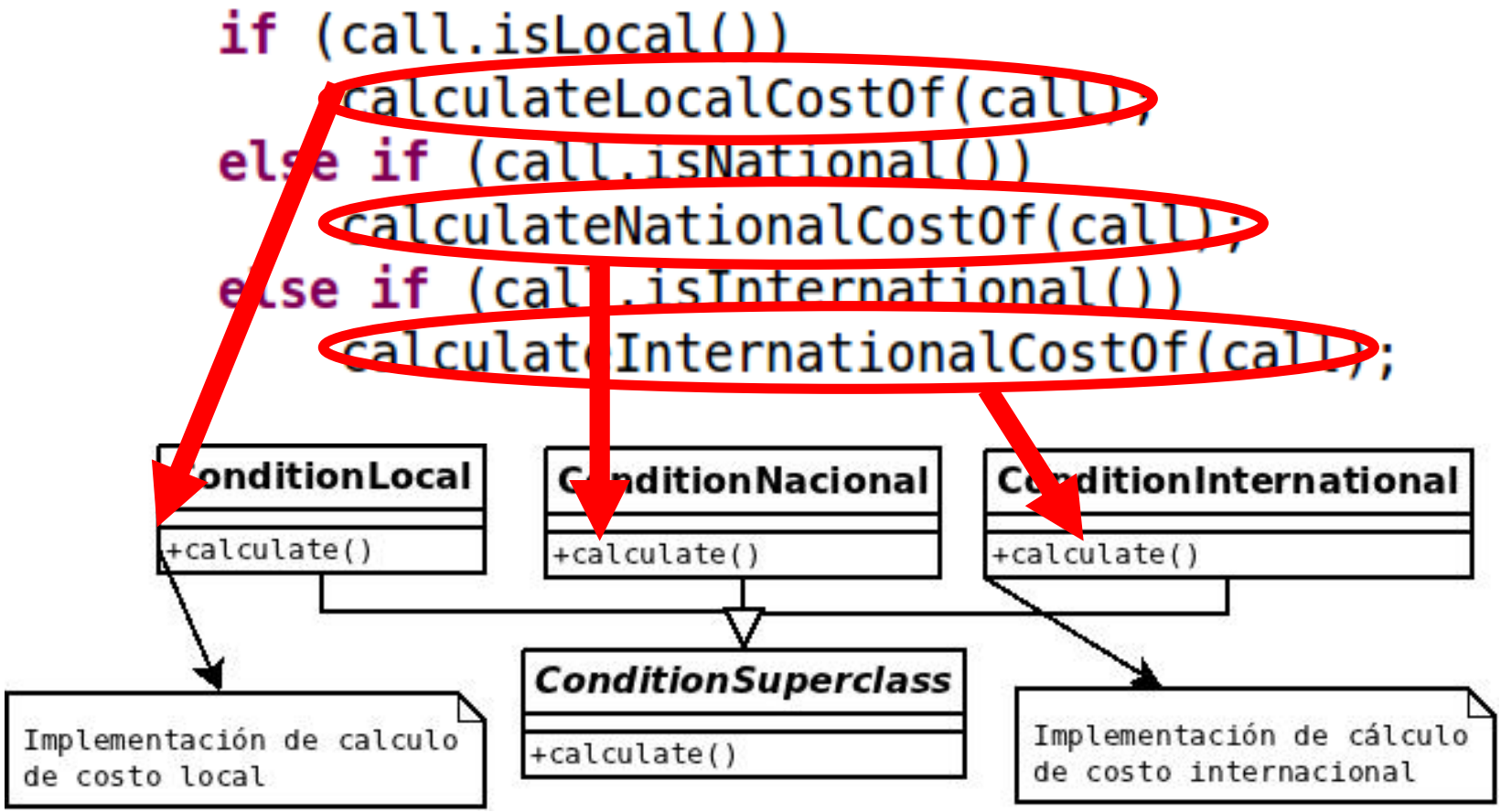
If – Cómo sacarlo

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



If – Cómo sacarlo

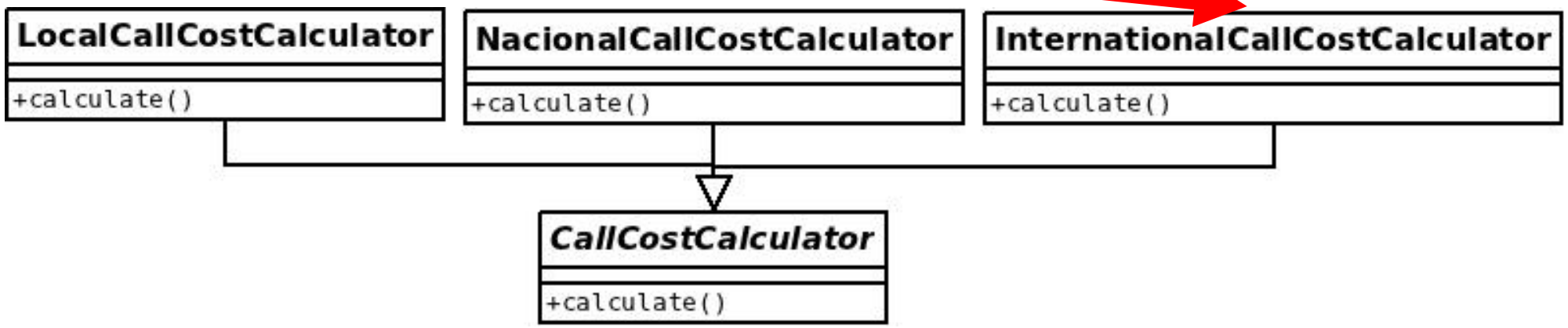
3. Nombrar el mensaje del paso anterior



If – Cómo sacarlo

4. Nombrar las abstracciones

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

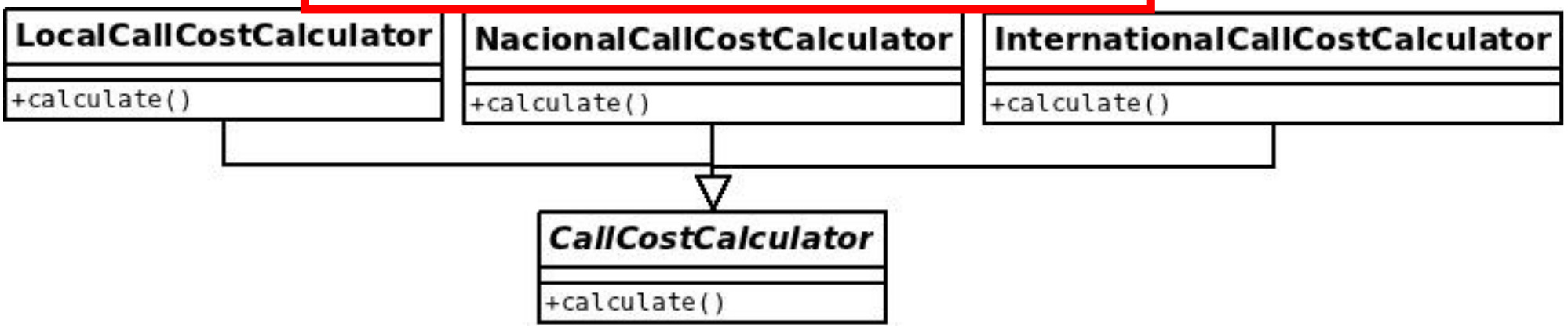


If – Cómo sacarlo

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();
```

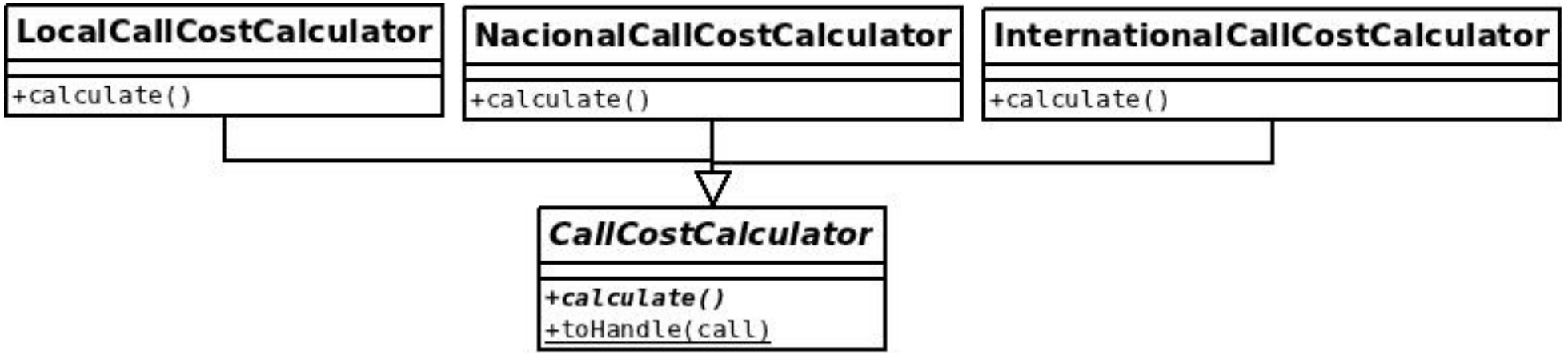


If – Cómo sacarlo

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 Heurística 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

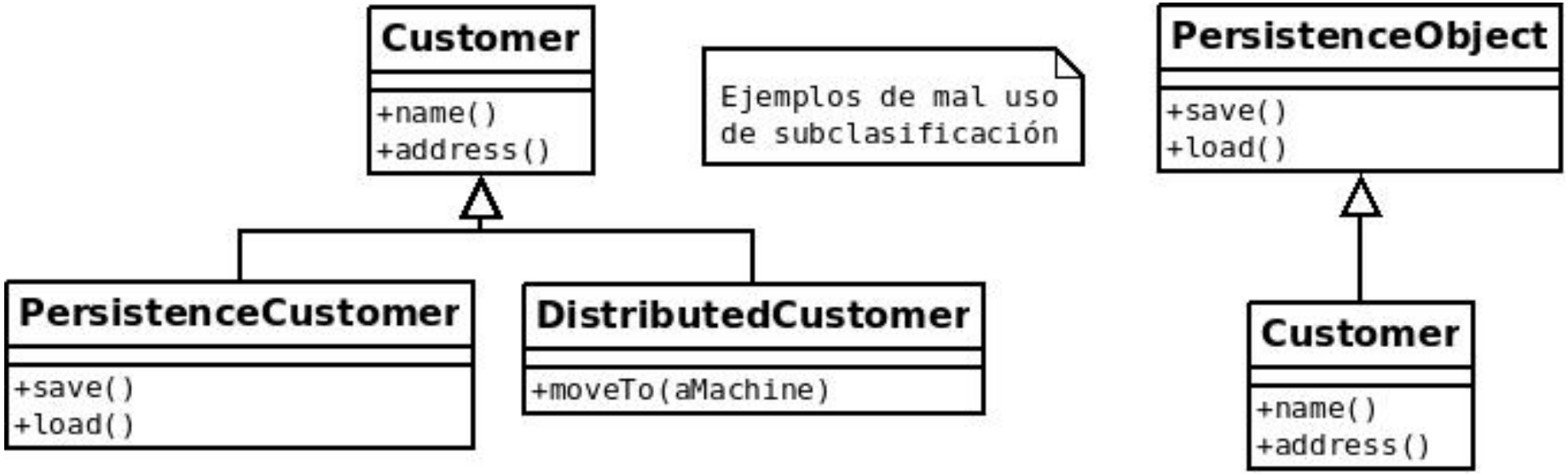
Heurísticas de Diseño

- **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*

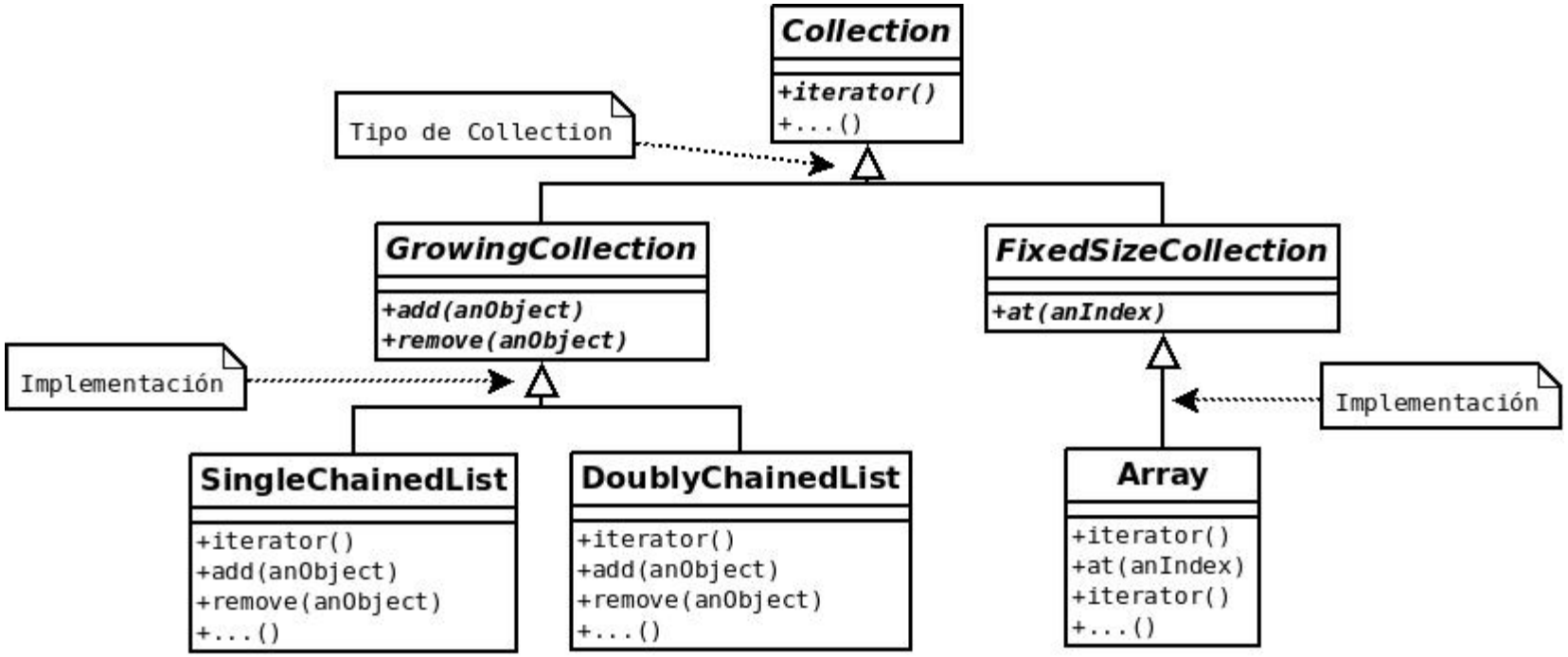
Heurísticas de Diseño

- **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

Software Robusto-Chequeo de Errores

- ¿Qué problemas tiene?

```
public Object m1 () {
    ...
    resultado = objeto.m2();
    if (resultado.isError()) return resultado;

    resultado = objeto.m3();
    if (resultado.isError()) return resultado;

    resultado = objeto.m4();
    if (resultado.isError()) return resultado;
    ...
}
```

Software Robusto-Chequeo de Errores

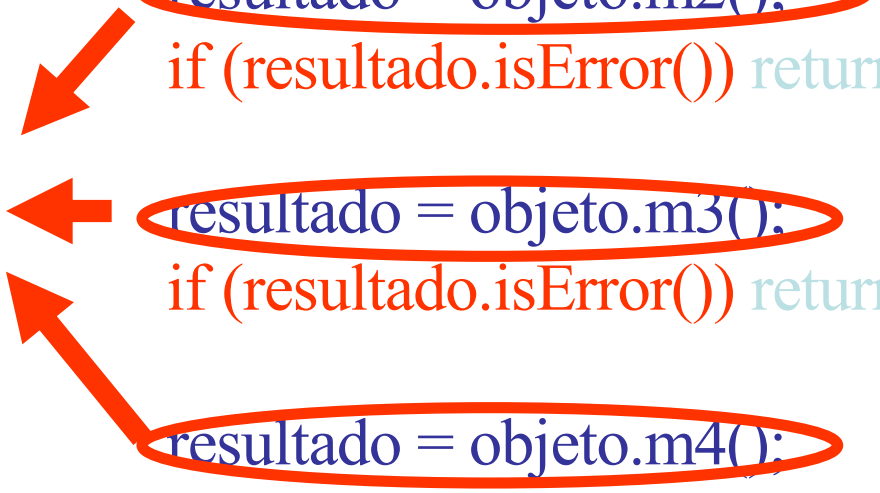
- Código repetido

```
public Object m1 () {
    ...
    resultado = objeto.m2();
    if (resultado.isError()) return resultado;

    resultado = objeto.m3();
    if (resultado.isError()) return resultado;

    resultado = objeto.m4();
    if (resultado.isError()) return resultado;
    ...
}
```

Qué se quiere hacer



Software Robusto-Chequeo de Errores

- Código Repetido

```
public Object m1 () {
    ...
    resultado = objeto.m2();
    if (resultado.isError()) return resultado;

    resultado = objeto.m3();
    if (resultado.isError()) return resultado;

    resultado = objeto.m4();
    if (resultado.isError()) return resultado;
    ...
}
```

Se verifica si hubo error

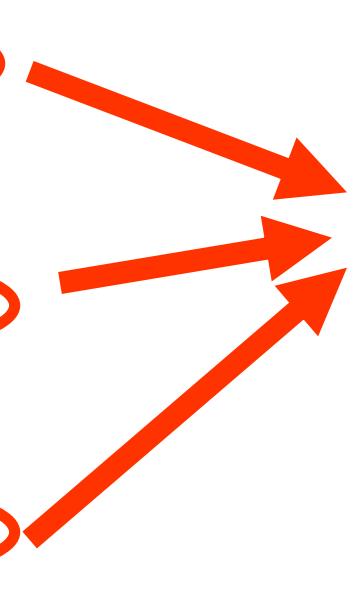
Software Robusto-Chequeo de Errores

- Código Repetido

```
public Object m1 () {
    ...
    resultado = objeto.m2();
    if (resultado.isError()) return resultado;

    resultado = objeto.m3();
    if (resultado.isError()) return resultado;

    resultado = objeto.m4();
    if (resultado.isError()) return resultado;
    ...
}
```



Se “handlea” el error

Software Robusto-Chequeo de Errores

- Problemas
  - Técnica propensa a error
  - Dificulta la legibilidad
  - Hace que la ejecución sea más lenta
  - Es antinatural puesto que el ser humano no esta acostumbrado a pensar de antemano en casos erróneos
  - Código Repetido!! → FALTA UN OBJETO!!

Chequeo de Errores → Exceptions

- Saco código repetido

```
public Object m1 () {
    ...
    resultado = objeto.m2();
    if (resultado.isError()) return resultado;

    resultado = objeto.m3();
    if (resultado.isError()) return resultado;

    resultado = objeto.m4();
    if (resultado.isError()) return resultado;
    ...
}
```

```
public Object m1 () {
    INTENTAR {
    objeto.m2();
    objeto.m3();
    objeto.m4();
    } isError() {
      return resultado;
    }
}
```

Exceptions

- Modelo del dominio de problema de la ejecución que resuelve las falencias de la técnica de código de retorno
- No interviene en los objetos de negocio
- Implementado en un meta-nivel

Exceptions

- Clase de Excepción
  - Representa el hecho que se produjo una situación inesperada y que la misma es informada (no es cohesivo)
- Contexto de Ejecución
  - Encargado de indicar cómo se tratará la excepción (handling)
- Proceso
  - El encargado de buscar en el contexto de ejecución que tratará la excepción

Exceptions

- ¿Qué sucede cuando no se encuentra un handler para una excepción?
  - Depende de la implementación
    - Smalltalk delega en otro objeto que hacer
  - Lo lógico es que el Proceso decida que hacer y por lo tanto se pueda modificar su comportamiento

Exceptions

- Problemas de las implementaciones actuales:
  - Utilizar subclasificación para modelar los distintos tipos de excepciones, obliga a tener un único motivo de clasificación de excepciones
  - Implementación es cerrada (Java, .Net) y por lo tanto no se la puede ampliar o modificar

Exceptions

- ¿Qué se puede hacer cuando se produce una excepción?
  - Reintentar el conjunto de colaboraciones que generó la excepción
  - Retornar del conjunto de colaboraciones que genero la excepción
  - Continuar en la próxima colaboración a la que produjo la excepción
  - Pasar la excepción al próximo handler
  - Transformar la excepción en otra

Exceptions

- ¿Qué exception levantar?
  - Si somo “dueños” del código:
    - Levantar una exception común (ej. Error en Smalltalk, RuntimeException en Java, etc), con la descripción de error correspondiente
    - Sólo crear un nuevo tipo de exception si se la debe handlear explicitamente
  - Si es una librería/framework que no tendrá control sobre su uso:
    - De manera juiciosa, crear un tipo de exception por cada condición de error que supongamos los usuarios querrán handlear

Exceptions

- Sintaxis vs Objetos

```
public Object m1 () {
    try {
    objeto.m2();
    objeto.m3();
    objeto.m4();
    } catch (Error e) {
      // Handlear error
    }
}
```

```
m1
    [ objeto m2.
    objeto m3.
    objeto m4 ]
    on: Error
    do: [ :e | Handlear error ]
```

(Smalltalk)

Contratos

- Definen en qué situación se puede realizar una colaboración y qué se espera como resultado de realizarla
- Están definidos en:
  - Pre-condiciones: Se evaluan para asegurarse que un método puede ser ejecutado
  - Pos-condiciones: Se evaluan al terminar un método para asegurarse que pasó lo que debería pasar
  - Invariantes: Son las condiciones que aseguran la consistencia de cada instancia

Contratos

- Pre-Condiciones:
  - Implementarlas SIEMPRE como aserciones al principio de cada método, por ej. para asegurar objetos válidos en mensaje de construcción de instancia
  - Dos políticas:
    - C/Berkeley: Objeto emisor asegura que se cumplan las pre-condiciones
    - Lisp/MIT: Objeto receptor asegura que se cumplan las pre-condiciones
  - Usar política Lisp/MIT si queremos software robusto y Fail-Fast

Contratos

- Post-Condiciones e Invariantes
  - Se pueden definir explicitamente pero conllevan mucho procesamiento y la necesidad de predicar sobre objetos anteriores al inicio de la ejecución del método
  - Otra opción: definirlas en tests automatizados. Esta es la opción que tomaremos nosotros