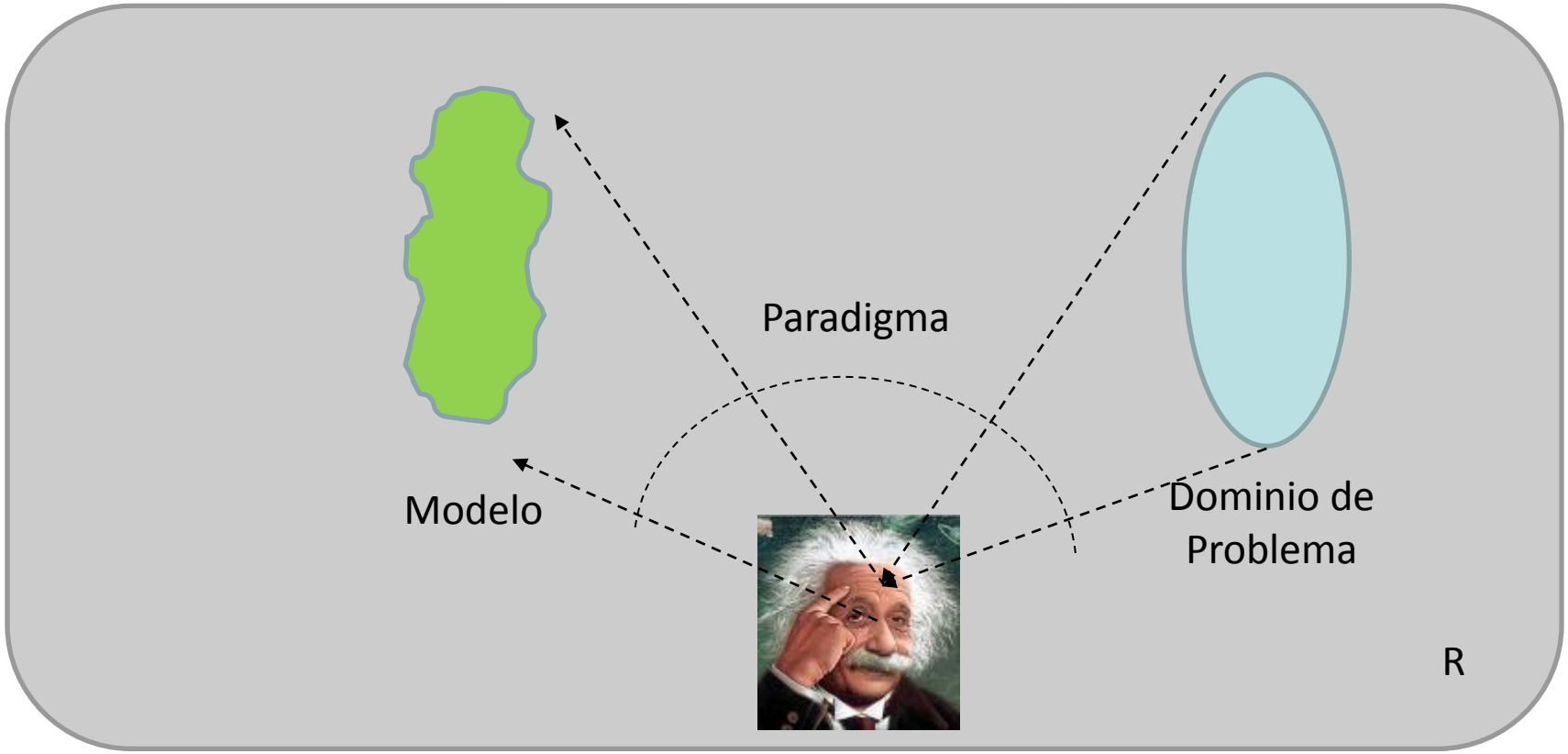
Ingeniería de Software II
UBA - FCEyN

Hernán Wilkinson

Twitter: @HernanWilkinson
Blog: objectmodels.blogspot.com
www.10pines.com

Definición de Software

Modelo Computable de un **Dominio de Problema** de la **Realidad**



Definición de Software

- **Realidad**: Todo aquello que podemos percibir, tocar, hablar sobre, etc

- **Dominio de Problema**: Un recorte de la realidad que nos interesa para el negocio que estamos modelando

Definición de Software

- **Modelo**: Representación de aquello que se está modelando

- **Computable**: Que puede ejecutar en una máquina de Turing → Formal, a-contextual
  - Característica esencial: No solo especifica el qué sino que además implementa el cómo

http://confreaks.net/videos/282-lsrc2010-real-software-engineering

Modelo

- **Buen Modelo**:
  - Eje Funcional: Qué tan buena es la representación del dominio
  - Eje Descriptivo: Qué tan bien está descripto el modelo, qué tan “entendible es”
  - Eje Implementativo: Cómo “ejecuta” en el ambiente técnico

Buen Modelo – Eje Funcional

- Un modelo es bueno cuando puede representar correctamente toda observación de aquello que modela
  - Si aparece algo nuevo en el dominio, debe aparecer algo nuevo en el modelo (no modificarlo)
  - Si se modifica algo del dominio, solo se debe modificar su representación en el modelo
  - Relación 1:1 dominio-modelo (isomorfismo)
  - Es la parte "**observacional**" del desarrollo

Buen Modelo – Eje Descriptivo

- Un modelo es bueno cuando se lo puede “entender” y por lo tanto “cambiar”
  - Importantísimo usar buenos nombres
  - Importantísimo usar mismo lenguaje que el del dominio de problema
  - El código debe ser “lindo”
  - Es la parte “**artística**” del desarrollo

http://www.10pines.com/content/art-naming

http://www.10pines.com/content/about-names-when-designing-objects

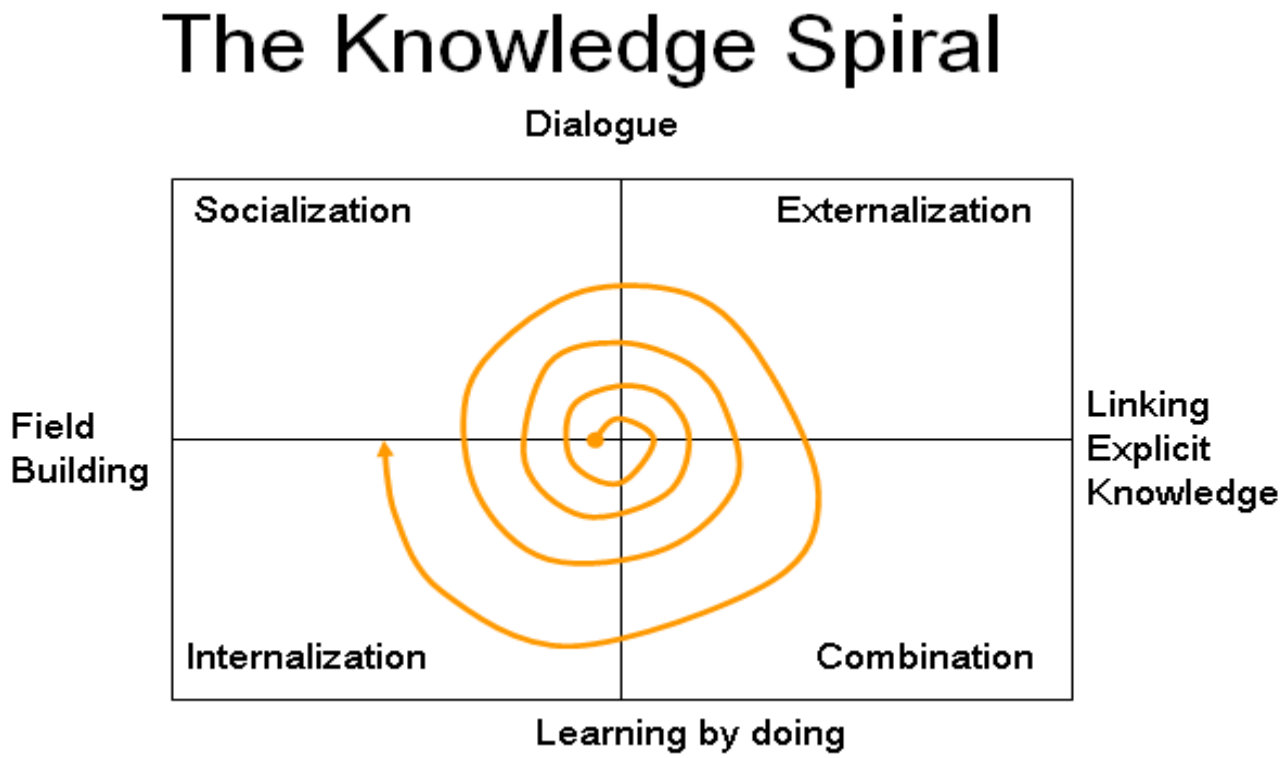
Buen Modelo – Eje Implementativo

- Un modelo es bueno cuando ejecuta en el tiempo esperado usando los recursos definidos como necesarios
  - Performance
  - Espacio
  - Escalabilidad
  - Todo lo relacionado con Requerimientos No Funcionales
  - Es la parte “**detallista**” del desarrollo

Proceso de Desarrollo de Software

- El desarrollo de software es un **proceso de aprendizaje**, lo que implica:
  - Es iterativo
  - Es incremental
  - El conocimiento se genera a partir de hechos concretos
  - El conocimiento generado debe ser organizado

Adquisición de Conocimiento

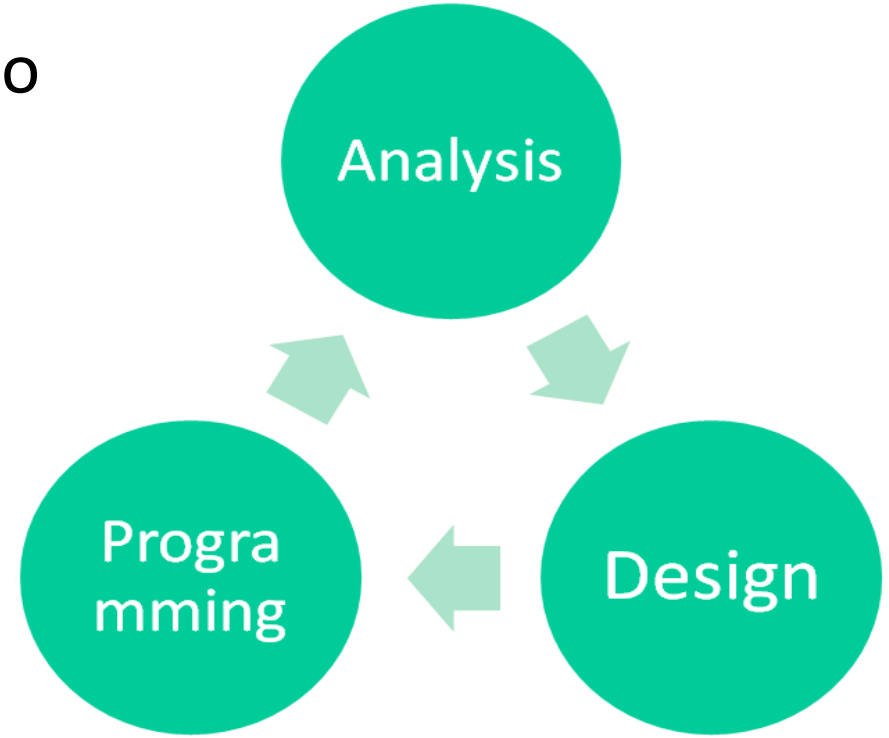


- Necesitamos Convertir
  - Conocimiento Implícito → Conocimiento Explícito
  - Conocimiento Informal → Conocimiento Formal

(Ver. L.A. Miller (1970),Natural Language programming)

Representación de Conocimiento

Lenguaje Natural
- Informal
- Incompleto
- Tácito



Lenguaje de Programación
- Formal
- Completo
- Explícito

Diagramas
- Informal
- Incompleto
- Explícito

Proceso de Desarrollo de Software

- Características
  - El dominio de problema está generalmente especificado en lenguajes ambiguos y contextuales (ej. Lenguaje natural)
  - El proceso de desarrollo implica desambiguar y descontextualizar el conocimiento del dominio de problema

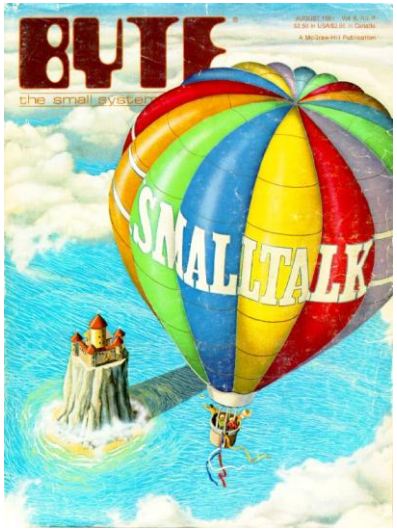
Proceso de Desarrollo de Software

- Características
  - El proceso de desarrollo implica hacer explícito y externo el conocimiento implícito e internalizado de los expertos de dominio
  - **El CAMBIO** es una característica esencial del software, no accidental porque:
    - Cambia el dominio de problema
    - Cambia nuestro entendimiento del dominio de problema
    - Cambia la manera de modelar lo que entendemos del dominio de problema

Paradigma de O

Dahl and Nygaard at the time of Simula's development

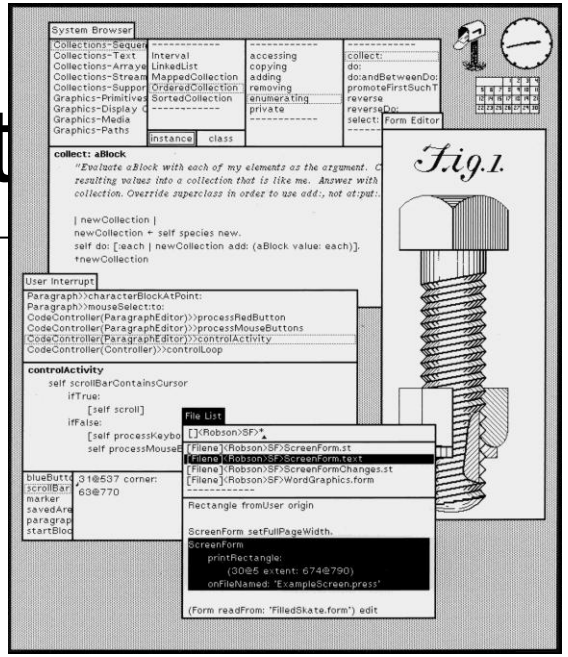
- Historia
  - Simula 67 (Nygrard y Dahl)
    - Antes que paradigma Estrucuturado
    - Goto Considered Harmfull – 68
    - Structured Programming – 71
      (using Simula 67 as prog. lang.!!)
  - Smalltalk
    - Alan Kay
    - Dan Ingalls
    - Adele Goldberg



Paradigma de Objet

DynaBook

Augment Children Comprehention



GUI - IDE

Smalltalk (72,74,76,78,80)

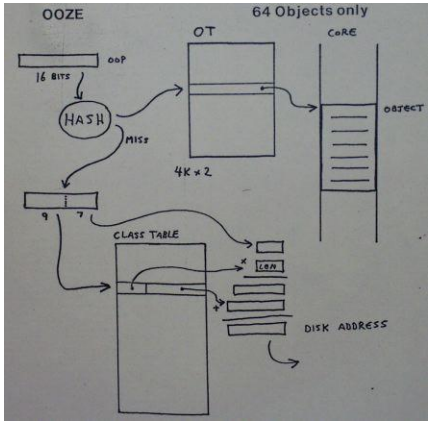
Object Oriented

Simula 67

Lisp

Flex Machine

VM



http://www.youtube.com/watch?v=AuXCc7WSczM

Paradigma de Objetos

- Alan Kay
  - "The best way to predict the future is to invent it"
  - "I invented the term Object-Oriented and I can tell you I did not have C++ in mind.“
  - "Java and C++ make you think that the new ideas are like the old ones. Java is the most distressing thing to hit computing since MS-DOS.“

http://video.google.com/videoplay?docid=-2950949730059754521

Paradigma de Objetos

- Principios Originales:
  - Simplicidad
  - Consistencia
  - Concretitud
  - Inmediate Feedback
  - Hacer Frente a la Complejidad

http://vimeo.com/36579366

Paradigma de Objetos

- Paradigma:
  - Definición minimalista
    - Conjunto de axiomas básicos mínimos
    - Pensar mucho antes de agregar un nuevo axioma
  - Puro
    - No traer conceptos de otros paradigmas, crearlos en base al paradigma
  - Actitud “Fundamentalista”
    - No salir de nuestra actitud minimalista/purista
- Liberarnos del lenguaje
- Liberarnos de otros paradigmas

Paradigma de Objetos

- Programa: **Objetos** que **Colaboran** entre si enviándose **mensajes**

Paradigma de Objetos

- **Objeto**: Representación de un ente del dominio de problema
  - No es código + datos! (error de definición)
  - **Ente**: Cualquier cosa que podamos observar, hablar sobre, etc.
  - La **esencia** del ente es modelado por los **mensajes** que el objeto sabe responder

Paradigma de Objetos

- **Mensaje**: Especificación sobre QUE puede hacer un objeto
  - Un Mensaje es un objeto!
  - Por lo tanto representa un ente de la realidad, pero del dominio de la “comunicación”
  - Se debería poder decir **qué** representa un objeto a partir de los mensajes que sabe responder

Paradigma de Objetos

- **Colaboración**: Hecho por el cual dos objetos se comunican por medio de un mensaje
  - En toda colaboración existe:
    - Un emisor del mensaje
    - Un receptor del mensaje
    - Un conjunto de objetos que forman parte del mensaje (colaboradores externos o parámetros)
    - Una respuesta

Paradigma de Objetos

- Características de las colaboraciones:
  - Dirigida (no broadcast)
  - Son sincrónicas, el emisor no continúa hasta que obtenga una respuesta del receptor
  - El receptor desconoce al emisor → su reacción será siempre la misma no importa quién le envía el mensaje
  - Siempre hay respuesta
- ¿Qué pasa si se cuestiona alguna de estas características?
  - Subjectivity (context aware…)

Paradigma de Objetos

- **Método**
  - Objeto que representa un conjunto de colaboraciones
  - Es evaluado como el resultado de la recepción de un mensaje por parte de un objeto
  - El método se busca utilizando el algoritmo de *Method Lookup*
  - Mismo mensaje principal que un programa: *execute* o *value*

Paradigma de Objetos

- **Relación de Conocimiento**
  - Es la única relación que existe entre objetos
  - Es la que permite que un objeto colabore con otro

Paradigma de Objetos

- **Variable**
  - Nombre contextual que se le asigna a una relación de conocimiento
  - Implica que el objeto conocido es llamado, en el contexto del "*conocedor*", de acuerdo a dicho nombre
  - No tiene ninguna implicación respecto del lugar que ocupa en memoria, cuánto ocupa, etc.

Paradigma de Objetos

- **Pseudo Variable “self” o “this”**
  - Nombre que referencia al objeto que está evaluando el método

- Características de las pseudo-variables:
  - No deben ser definidas
  - No pueden ser asignadas

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 ].
```

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

```
aClosure value -> 3628800.
```

… y se le pueden enviar mensajes

```
| aClosure |

aClosure := [ :anInteger | anInteger factorial ].
```

Puede recibir “colaboradores”

```
aClosure value:10 -> 3628800.
```

… 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 – C#

Declaro el tipo

```
delegate int integerLambda(int anInteger);

public void m1()
{
    integerLambda addFive = anInteger => anInteger + 5;

    addFive(10); //Retorna 15
    addFive.Invoke(10); //Retorna 15
}
```

Defino la expr. lambda

Lo uso

Closure + Generics – C#

```
delegate TResult genericOperation<in T, out TResult>(T arg);

public void m1()
{
    genericOperation<int,int> addIntegerFive= anInteger => anInteger + 5;
    addIntegerFive(10); //Retorna 15

    genericOperation<float,float> addFloatFive= aFloat => aFloat + 5;
    addFloatFive(10); //Retorna 15.0

}
```

Lambda expressions son Closure – C#

```
public static Func<int, int> m1()
{
    int leftOperand = 10;

    return anInteger => leftOperand = leftOperand + anInteger;
}

public static void Main(string[] args)
{
    Func<int, int> adder = m1();
    Console.WriteLine(adder(5)); // Retorna 15
    Console.WriteLine(adder(5)); // Retorna 20
```

Closure - Java

- ¡No tiene!
- Se utilizan clases anónimas…
- Ejemplo:

```
public interface NoArgumentClosure<T> {
    public T execute();
}
```

Clase anónima - Java

```
public interface OneArgumentClosure<ReturnType,ArgumentType> {
    public ReturnType execute(ArgumentType firstArgument);
}


public static void main (String[] args){
    OneArgumentClosure<Integer,Integer> addFive =
        new OneArgumentClosure<Integer,Integer>() {
        public Integer execute (Integer anInteger) { return anInteger + 5; }
    };

    addFive.execute(10);  // retorna 15
}
```

Clase anónima - Java

- ¿Por qué no es closure?

```
public static OneArgumentClosure<Integer,Integer> m1() {

   int leftOperand = 10;

   return new OneArgumentClosure<Integer,Integer>() {
      public Integer execute (Integer anInteger) {
         leftOperand = leftOperand + 5;
         return leftOperand; }
   };
}
```

Cannot refer to a non final variable…

Clase anónima - Java

- ¿Por qué no es closure?

```
public static OneArgumentClosure<Integer,Integer> m1() {

    final int leftOperand = 10;

    return new OneArgumentClosure<Integer,Integer>() {
        public Integer execute (Integer anInteger) {
            leftOperand = leftOperand + 5;
            return leftOperand; }
    };
}
```

The final local variable leftOperand cannot be assigned

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);
```

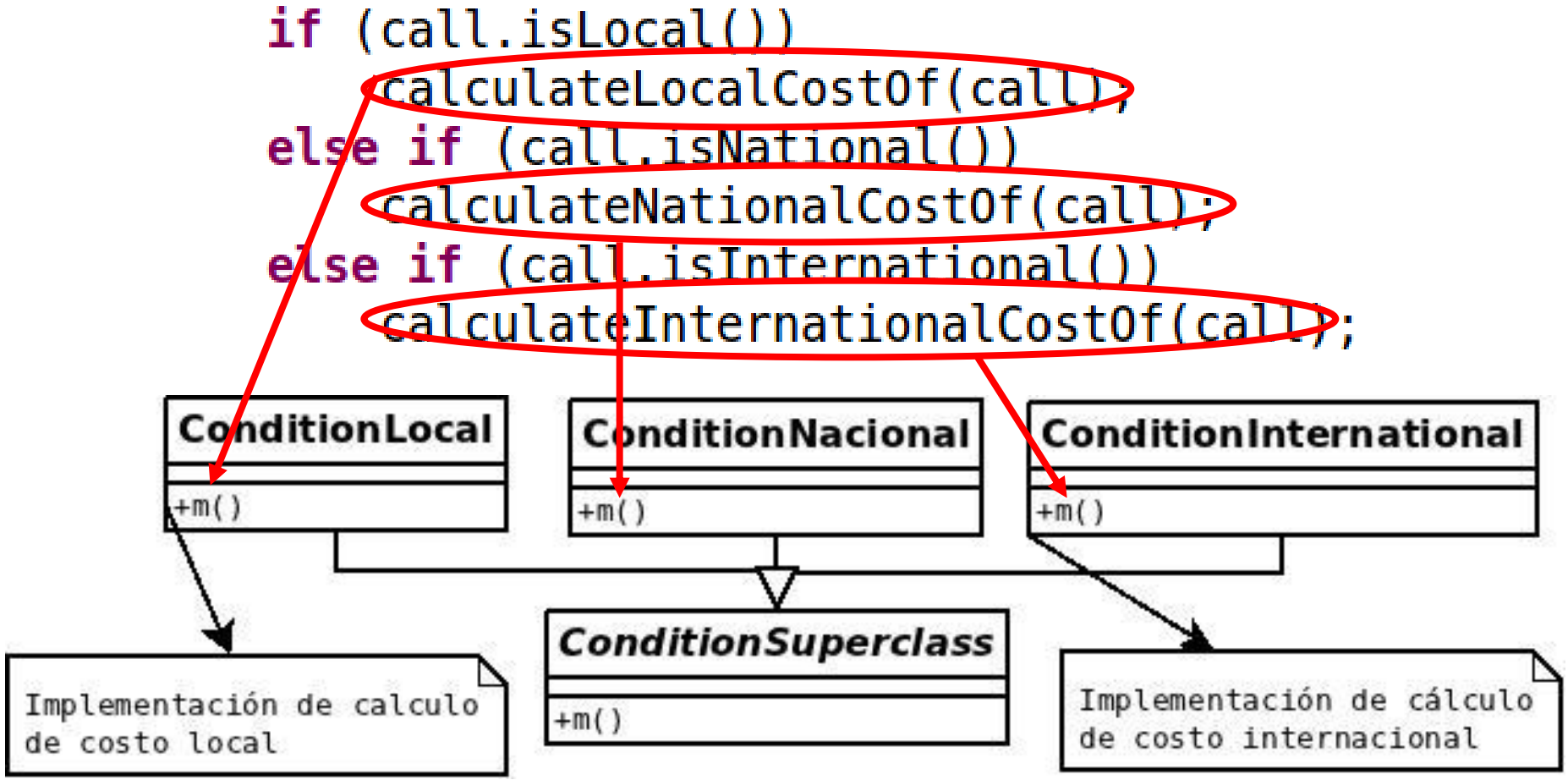
If – Cómo sacarlo

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

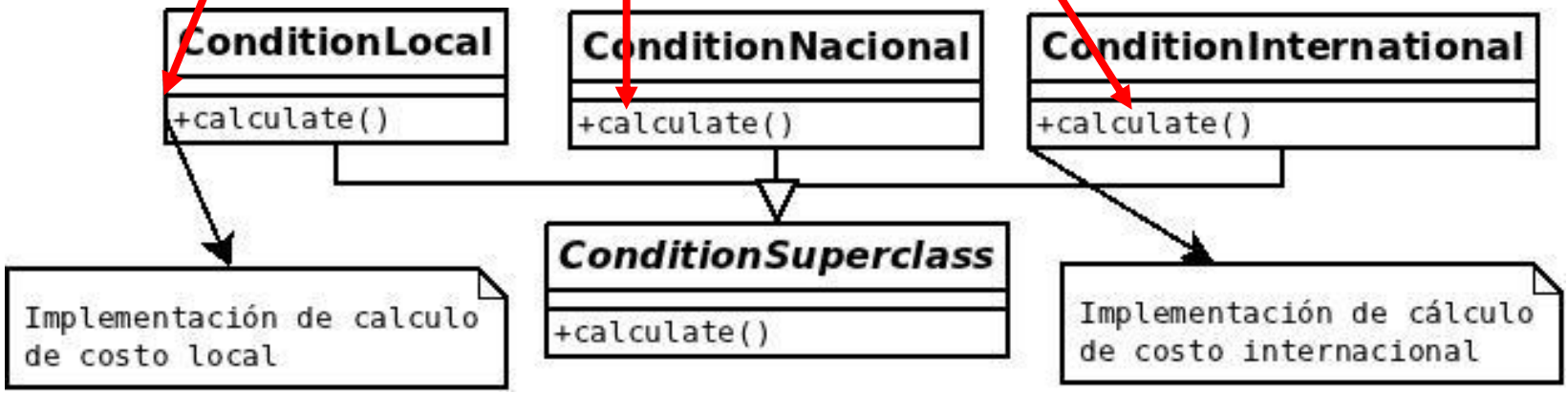
2. Usando el mismo nombre de mensaje repartir el "cuerpo del if" en cada abstracción



If – Cómo sacarlo

3. Nombrar el mensaje del paso anterior

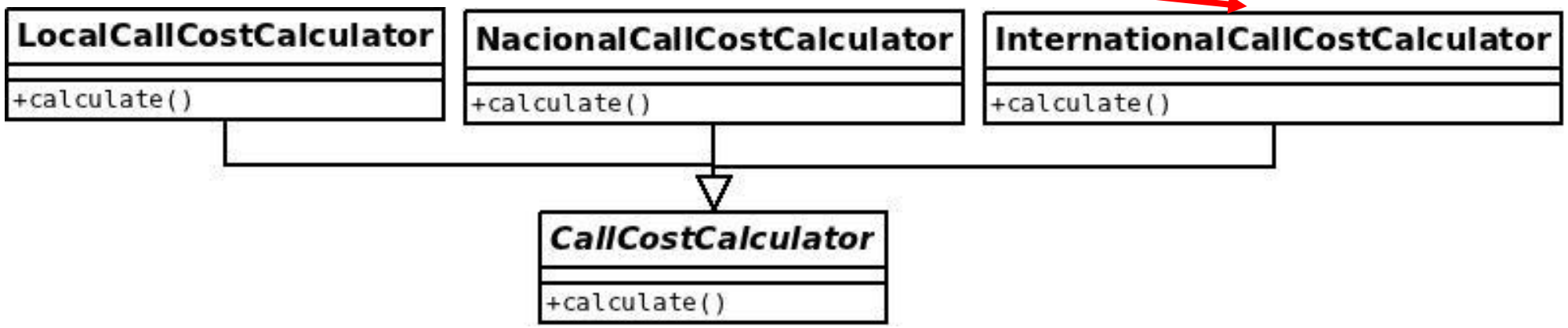
```
if (call.isLocal())
    calculateLocalCostOf(call);
else if (call.isNational())
    calculateNationalCostOf(call);
else if (call.isInternational())
    calculateInternationalCostOf(call);
```



If – Cómo sacarlo

4. Nombrar las abstracciones

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

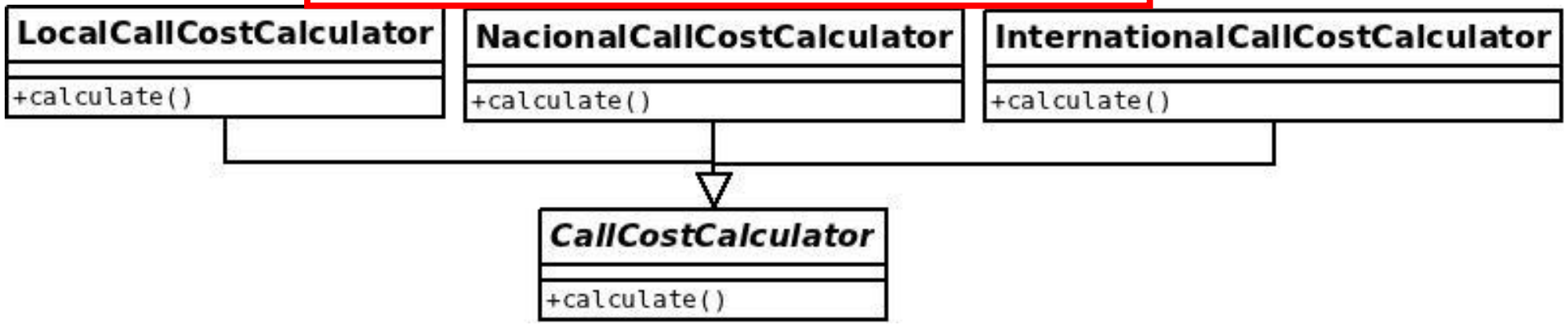


If – Cómo sacarlo

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();
```

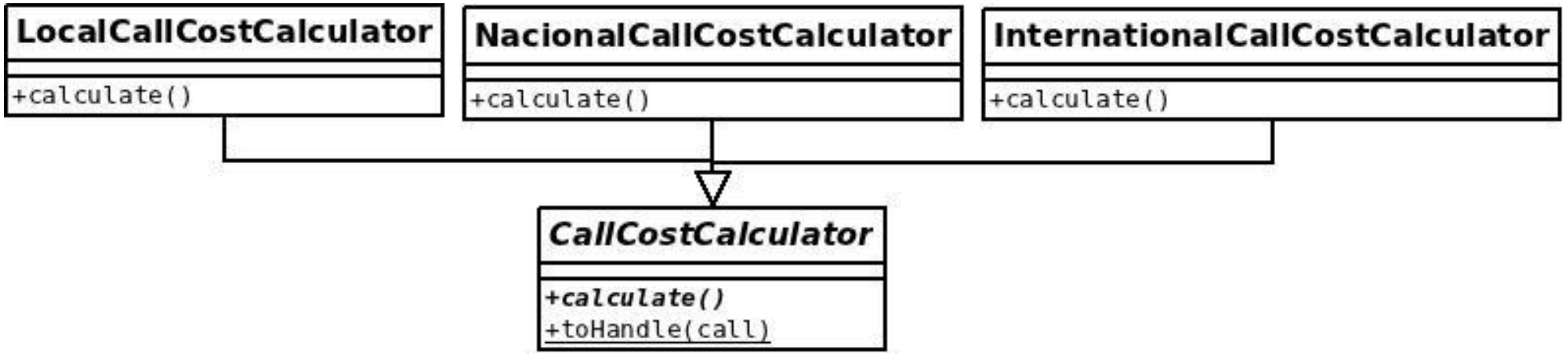


If – Cómo sacarlo

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…

Reglas (o heurísticas) de Diseño

- **Regla 1**: 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

Reglas (o heurísticas) de Diseño

- **Regla 2**: Los objetos deben ser cohesivos representando responsabilidades de un solo dominio de problema

- Cuanto más cohesivo es un objeto más reutilizable es

Reglas (o heurísticas) de Diseño

- **Regla 3**: 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

Reglas (o heurísticas) de Diseño

- **Regla 4**: 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

Reglas (o heurísticas) de Diseño

- **Regla 5**: Se deben utilizar clases abstractas para representar conceptos abstractos

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

Reglas (o heurísticas) de Diseño

- **Regla 6**: 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

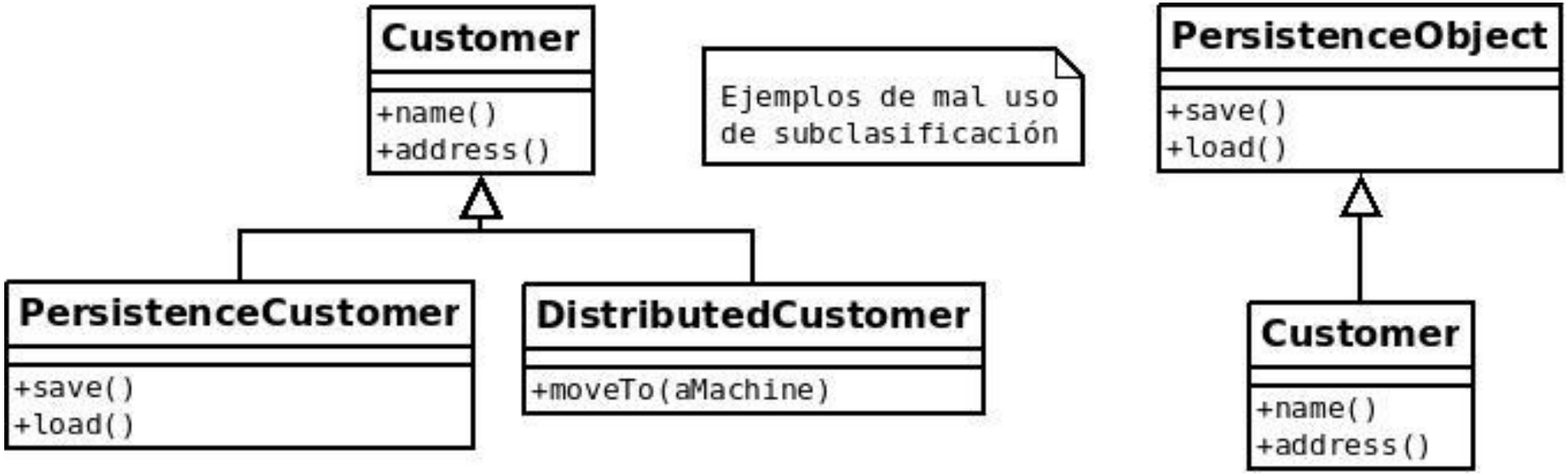
Reglas (o heurísticas) de Diseño

- **Regla 7**: 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*

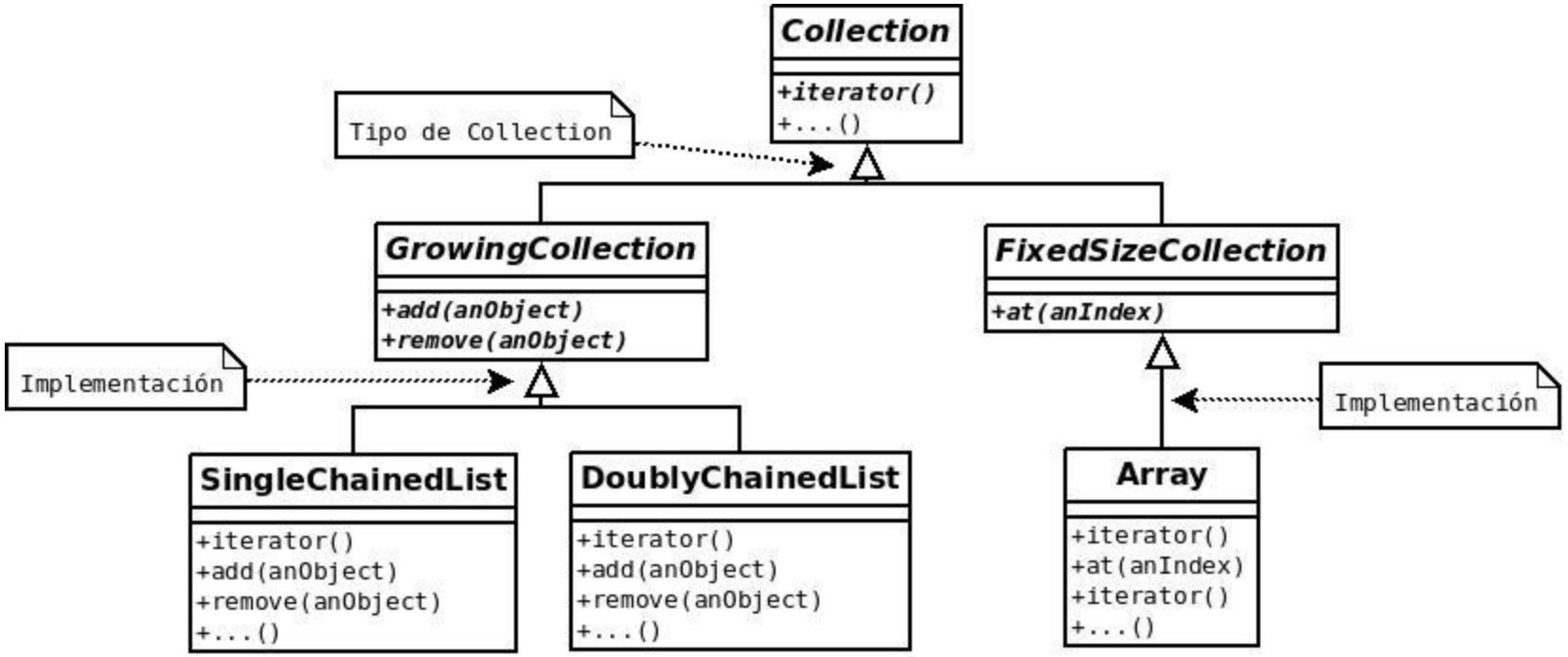
Reglas (o heurísticas) de Diseño

- **Regla 8**: El motivo de subclasificación debe pertenecer al dominio de problema que se esta modelando



Reglas (o heurísticas) de Diseño

- **<u>Regla 9</u>**: No se deben mezclar motivos de subclasificación al subclasificar una clase



Reglas (o heurísticas) de Diseño

- **Regla 10**: 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

Reglas (o heurísticas) de Diseño

Regla 11: 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

Reglas (o heurísticas) de Diseño

- **Regla 12**: 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

Reglas (o heurísticas) de Diseño

- **Regla 13**: 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

Reglas (o heurísticas) de Diseño

- **Regla 14**: 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*

Reglas (o heurísticas) de Diseño

- **Regla 15**: 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

Reglas (o heurísticas) de Diseño

- **Regla 16**: 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)*

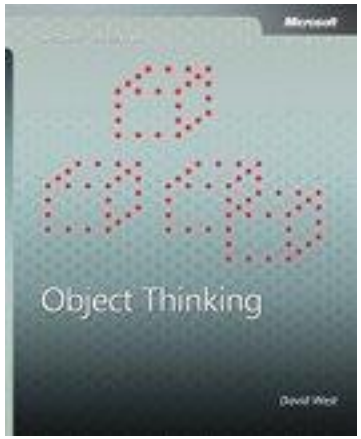
Reglas (o heurísticas) de Diseño

- **Regla 17**: Modelar la arquitectura del sistema

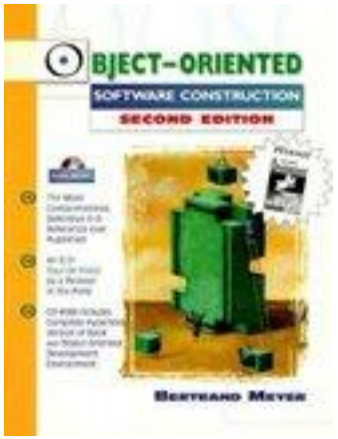
  - Crear un modelo de la arquitectura del sistema (subsistemas, etc)
  - Otorgar a los subsistemas la responsabilidad de mantener la validez de todo el sistema (la relación entre los objetos)
  - Otorgar la responsabilidad a los subsistemas de modificar un objeto por su impacto en el conjunto

Bibliografía y referencias

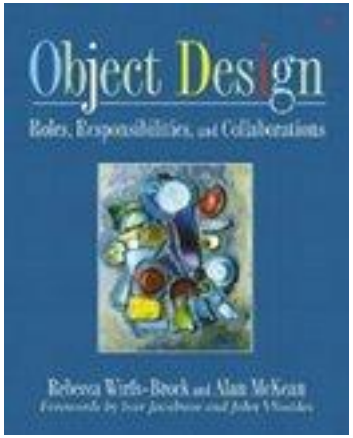
Bibliografía recomendada



Object Thinking.
David West, 2004.



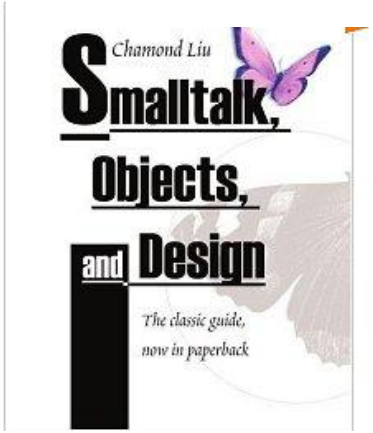
Object-Oriented Software Construction.
Bertrand Meyer, 2000.



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

Bibliografía recomendada

Smalltalk Best Practice Patterns. Kent Beck



Smalltalk, Objects and Design Chamond Lie