Ingeniería de Software II
Diseño con Objetos

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Autor: Hernán Wilkinson

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

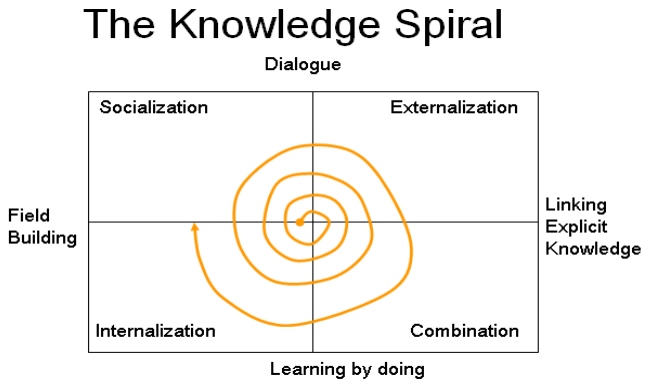
Modelo

- **Buen Modelo**: Un modelo es bueno cuando puede representar correctamente toda observación de aquello que modela
  - En software, por ser un modelo computable, la palabra *correctamente* tiene connotaciones relacionadas a características computables como performance, espacio, etc.

Proceso de Desarrollo de Software

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

Adquisición de Conocimiento

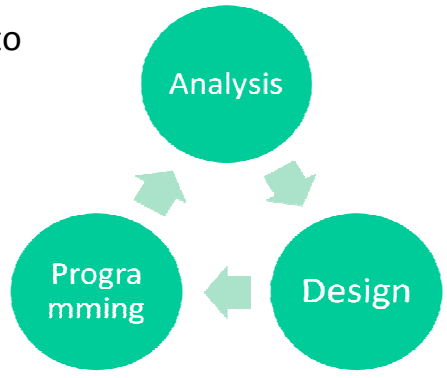


- Necesitamos Convertir
  - Conocimiento Implícito → Conocimiento Explícito
  - Conocimiento Informal → Conocimiento Formal

(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 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:
  - 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

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

- **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*

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 clasificación
  - 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

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

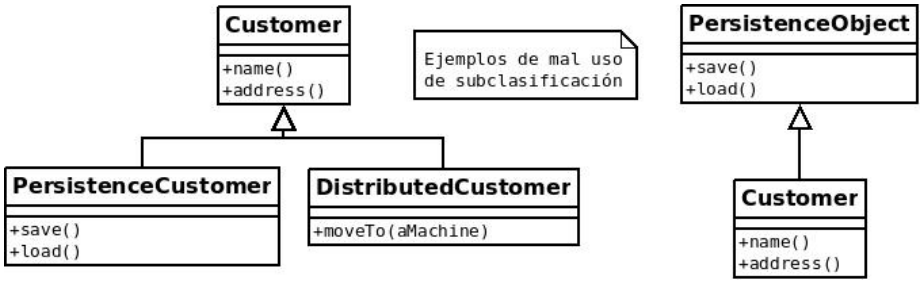
Reglas (o heurísticas) de Diseño

- **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*

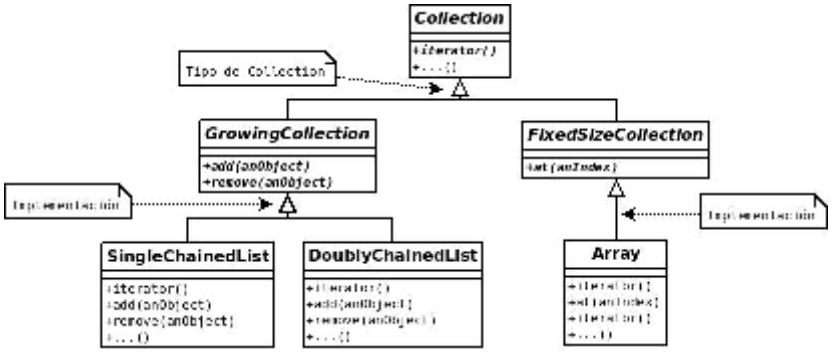
Reglas (o heurísticas) de Diseño

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



Reglas (o heurísticas) de Diseño

- **Regla 9**: 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

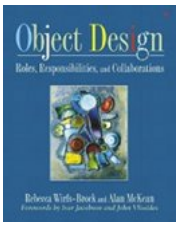
- Crear un modelo de la arquitectura del sistema (subsistemas, etc)
- Otorgar a los subsistemas la responsabilidad de mantener la validez de todo el sistema (la relación entre los objetos)
- Otorgar la responsabilidad a los subsistemas de modificar un objeto por su impacto en el conjunto

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