

# Algoritmos y Programacion III

Resumen de la materia Algoritmos y Programacion III - Catedra Suarez

Clase 1: Resolucion de Problemas con objetos

Método

Objeto

Encapsulamiento

Polimorfismo

Clase

Atributos

Recolección de Basura

Lenguaje Tipado vs Estático

Clase 2: Diseño por Contrato

Pruebas Unitarias

Atributos y Propiedades

Test-Driven Development

Paradigma OO y complejidad

Clase 3: Colaboraciones de objetos

Dependencia y delegación

Programación por diferencia

Delegación vs Herencia

Redefinición

Clases Abstractas

Herencia en Smalltalk y Java

Visibilidad

Inicialización y asociación

Clase 4: Polimorfismo y Refactorización

Soluciones Típicas

Interfaces

Refactorizacion

Clase 5: Excepciones y profundización de POO

¿Cuándo lanzar excepciones?

Jerarquía de Excepciones

Excepciones chequeadas en forma estática (Java)

Unified Modeling Language (UML)

Diagrama de Secuencia

Diagrama de Clases

Diagrama de Paquetes

Diagrama de Estado

Iterador

Clase 6: Cálidad de Código; XP

Legibilidad

Métodos

Comentarios

Mejora de Desempeño

Extreme Programming (XP)

Clase 7: Diseño Orientado a Objetos

Diseño

Modularización

**Cohesión y acoplamiento**

Diseño de Clases

Principios SOLID (Robert Martin)

**Delegación sobre herencia**

Diseño de Paquetes

**¿Cuándo crear métodos?**

Visibilidad

Clase 8: Metodologías de Desarrollo de Software

Desarrollo de Software

Características del Software

Fracasos del desarrollo de software

Algunas reflexiones

Metodología

Ciclos de vida de desarrollo de software

Lineal

Incremental

Categorías de Métodos

Manifiesto Ágil

Algunos Métodos Ágiles

Clase 9: RTTI y reflexión

RTTI

Reflexión

**En Java:**

**En Smalltalk:**

Java

Objetivos

Smalltalk

Clase Complementaria: Presentación MVC

Patrón Observer

Clase Complementaria: Persistencia

Serialización

### **Formatos de serialización**

Persistencia y serialización

Persistencia No Nativa

Clase Complementaria: Conurrencia

Conurrencia vs Paralelismo

Clase Complementaria: Interfaces de Usuario

Human-computer Interaction / HCI / CHI

Usabilidad

### **Gamificación o Idificación**

User Experience UX

Accesibilidad

Etapas en el diseño centrado en el usuario

Nielsen's Heuristics

Recomendaciones

Resumen de autor: Patrones de Diseño

Patrones Creacionales

1. SINGLETON

2. MULTITON

3. ABSTRACT FACTORY

Patrones de Organización

4. COMMAND

Patrones de Control de Acceso

5. PROXY

6. FACADE

Patrones de Variaciones de Servicios

7. STRATEGY

8. TEMPLATE

9. STATE

Patrones de Extensión de Servicios

10. DECORATOR

Patrones de Descomposición Estructural

11. COMPOSITE

# Clase 1: Resolucion de Problemas con objetos

Resolviendo con objetos:

1. Encontrar entidades del dominio del problema (*los objetos*)
2. Descubrir y establecer como interactuan esas entidades (*buscamos los mensajes entre objetos*)
3. Determinar cómo hacen esos objetos para responder a los mensajes (*comportamiento*)

## Programa OO

- Conjunto de objetos enviando mensajes a otros objetos.
- Los objetos receptores reciben los mensajes y reaccionan.
  - Haciendo algo (*comportamiento*).
  - Devolviendo un valor (*que depende de su estado*).
- Los mensajes pueden implicar la creación de nuevos objetos.
- El comportamiento puede delegarse a su vez en otro objeto.

## Método

Llamamos método a la implementacion de la respuesta de un objeto a un mensaje. Se asemeja a funciones o procedimientos de programación en otros paradigmas.

tablero >> puedoColocar(numero, filaCelda, columnaCelda)

## Objeto

Los objetos tienen **identidad**: es lo que lo distingue a un objeto de otro.

Los objetos tienen **estado**: tiene que ver con algo que cambia a través del tiempo.

Los objetos tienen **comportamiento**: conjunto de posibles respuestas de un objeto ante los mensajes que recibe.

## Encapsulamiento

Cada objeto es responsable de responder a los mensajes que recibe, sin que quien le envia el mensaje tenga que saber cómo lo hace

**“Tell, don’t ask”**

Los objetos deben manejar su propio comportamiento, sin que manipulemos su estado desde afuera.

## Polimorfismo

Capacidad de respuesta que tienen distintos objetos de responder de maneras diferentes a un mismo mensaje.

## Clase

**Clase** es un conjunto de objetos que, por lo menos, tienen el mismo comportamiento

- Todos los objetos de una clase entienden los mismos mensajes
- Todos los objetos de una clase responden a los mensajes de la misma manera

La clase es un tipo de objeto definido por el programador.

## Atributos

Es una variable interna del objeto que sirve para almacenar parte del estado del mismo contenido de Celda

## Recolección de Basura

Smalltalk, JavaScript y Java tienen mecanismos de recolección de basura.

- No es deterministica.
- Asegura que
  - No va a faltar memoria mientras haya objetos sin referencia.
  - No se va a liberar ningún objeto que esté siendo referenciado desde un objeto referenciado.

- Evita errores muy difíciles de encontrar y reparar.

## Lenguaje Tipado vs Estático

Los **lenguajes de tipado dinámico** son aquellos (como JavaScript) donde el intérprete asigna a las variables un tipo durante el tiempo de ejecución basado en su valor en ese momento.

Se dice de un **lenguaje** de programación que usa un **tipado estático** cuando la comprobación de tipificación se realiza durante la compilación, y no durante la ejecución.

## Clase 2: Diseño por Contrato

Es una forma de especificar el comportamiento de un objeto.

*Betrand Meyer: un objeto brinda servicios a sus clientes cumpliendo un contrato.*

4 elementos:

1. Firmas de métodos

Determinan cómo hacer para pedirles servicios a los objetos.

2. Precondiciones

Expresan en qué estado debe estar el medio antes de que un objeto reciba un mensaje.

*ej: Para que se pueda depositar en una cuenta el monto a depositar debe ser un número positivo.*

Si una precondición no se cumple, ocurre una **excepción**: es un objeto que el receptor de un mensaje envía a su cliente como aviso de que el propio cliente no está cumpliendo con alguna precondición.

3. Postcondiciones

El conjunto de postcondiciones expresa el estado en que debe quedar el medio como consecuencia de la ejecución de un método. En términos operativos, es la respuesta ante la recepción del mensaje.

Precondición: responsabilidad del cliente

Postcondición: responsabilidad del programador del comportamiento del objeto receptor.

#### 4. Invariantes

Son condiciones que debe cumplir un objeto durante toda su existencia.

*ej: Cualquier objeto cuenta, o bien no tiene saldo, o su saldo es mayor que 0.*

Responsabilidad de todos los métodos de un objeto.

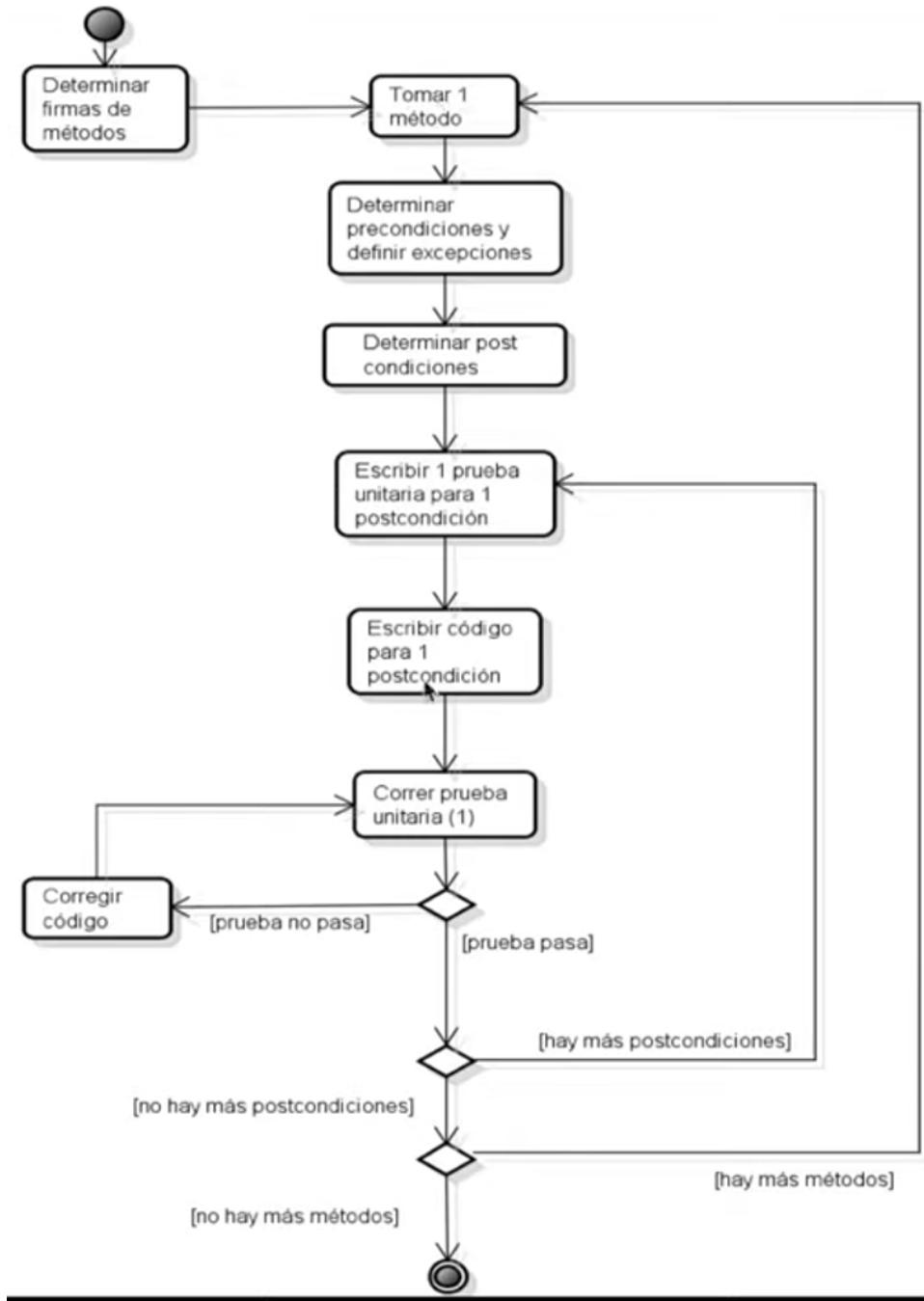
Suelen expresarse en forma de precondiciones o postcondiciones

## Pruebas Unitarias

Una prueba unitaria es aquella prueba que comprueba la corrección de una única responsabilidad de un método.

*Corolario: deberíamos tener al menos una prueba unitaria por cada postcondición.*

### Procedimiento básico



## Atributos y Propiedades

El **Atributo** es donde yo guardo el estado, mientras que la **Propiedad** es algo propio del objeto que no necesariamente esta en un atributo.

⇒ No todos los atributos tienen métodos de consulta y modificación.

⇒ Las propiedades deberían tener alguna forma de visibilidad.

Desarrollo que empieza por las pruebas

Ventajas:

Minimiza el condicionamiento del autor.

Las pruebas se convierten en especificaciones de lo que se espera como respuesta del objeto ante el envío de un mensaje.

Permite especificar el comportamiento sin restringirse a una implementación.

Deja un conjunto de pruebas de regresión.

Pruebas en código

Ventajas:

Permite independizarse del factor humano: menor subjetividad y variabilidad en el tiempo.

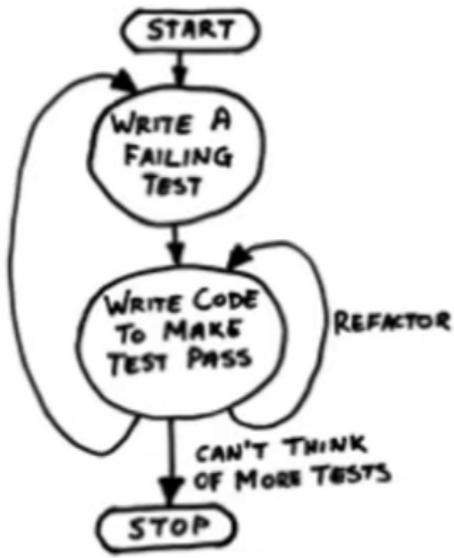
Facilita la repetición de las mismas pruebas, con un costo ínfimo: el programador las ejecuta con mayor frecuencia y regularidad.

## **Test-Driven Development**

Es una práctica interactiva-incremental de desarrollo de software.

Incluye

- Test-First
- Automatización
- Refactorización



Es un subproducto:

- Pruebas como ejemplos
  - Documentan
    - Firma del método
    - Posibles excepciones
    - Resultados que se esperan de su ejecución
  - Mejor que cualquier documento escrito en prosa
- Control de calidad
  - No es el objetivo con el que las hacemos
  - Sirven también como herramientas para los controles de regresión
  - Igual necesitaremos pruebas mas abarcativas

## Paradigma OO y complejidad

OO permite el manejo de la complejidad a un costo razonable.

→ construir en base a componentes (objetos).

Condiciones:

## 1. Encapsulamiento

Cada componente debe tener el comportamiento esperado sin que nuestro desarrollo dependa de la manera en que está implementado.

## 2. Contratos

Qué nos ofrece cada componente y como comunicarnos con el

### **Encapsulamiento + Contratos = Abstracción**

Claves

- Los objetos se implementan en base a un modelo cliente-proveedor
- De las excepciones deducimos las excepciones
- Con las postcondiciones escribimos pruebas unitarias
- Las pruebas unitarias guían la implementación de incrementos pequeños
- Hoy no sirve para manejar la complejidad y su auxiliar es la abstracción

## **Clase 3: Colaboraciones de objetos**

**Objeto receptor:** Aquél que recibe el mensaje en un escenario de interacción entre objetos. Brinda un servicio.

**Objeto Cliente:** Aquél que envía un mensaje en un escenario de interacción entre objetos. Consumo un servicio.

*obs: Todo mensaje tiene un objeto cliente y uno receptor.*

Colaboración por delegación: Se necesita alguna dependencia.

## **Dependencia y delegación**

Un objeto depende de otro cuando debe conocerlo para poder enviarle un mensaje.

Todo objeto cliente depende de su servidor ⇒ Decimos que el cliente delega en el receptor.

## Tipos de dependencias

1. El objeto receptor se envía como argumento
2. El objeto receptor se obtiene como respuesta al envío de un mensaje a otro objeto
3. El objeto cliente tiene una referencia al receptor ⇒ la **asociación**: Forma de dependencia en la que el cliente tiene almacenada una referencia al servidor.

## Programación por diferencia

Programamos por diferencia cuando indicamos que parte de la implementación de un objeto está definida en otro objeto, y por lo tanto sólo implementamos las diferencias específicas.

### Herencia (programación por diferencia)

Es una relación entre clases

Relación generalización - especialización:

- Tipo más genérico: ancestro, madre, base
- Tipo más específico: descendiente, hija, derivada

Si hay comportamiento en la clase madre, podemos utilizar ese comportamiento en cada clase hija.

También existe la **Herencia Múltiple**

*ojo: Las clases no parecen excluyentes, pero los objetos sólo pueden ser instancias de una clase.*

## Delegación vs Herencia

**Herencia relación “es un”:**

Toda instancia de una clase hija es instancia de la clase madre

**Composición/agregación:**

“contiene”, “hace referencia”, “es parte de”

Mito: *En POO todo es herencia*

### ¿Cuándo utilizar?

Herencia → cuando se va a reutilizar la interfaz tal como está.

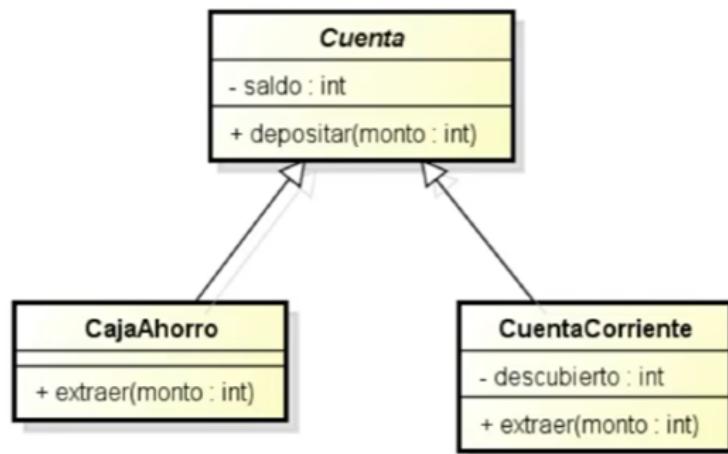
Delegación → Cuando se va a reutilizar sin mantener la interfaz.

## Redefinición

Para implementar de modo diferente la respuesta a un mismo mensaje

## Clases Abstractas

No pueden tener instancias en forma directa. Habitualmente porque sus descendientes cubren todos los casos posibles.



### Métodos abstractos

No lo implementamos en una clase. Pero deseamos que todas las clases descendientes puedan entender. *Ej: +extraer()*

## Herencia en Smalltalk y Java

## Smalltalk:

```
ColeccionCeldas subclass: #Fila
```

## Java:

```
class Fila extends ColeccionCeldas  
{ . . . }
```

## Visibilidad

Importante para garantizar el encapsulamiento

- Atributos y métodos **privados**: solo los puede usar el objeto receptor en su clase
- Atributos y métodos **públicos**: se los puede usar desde cualquier lado.
- Atributos y métodos **protegidos**: Sólo los puede usar el objeto receptor en su clase y en las clases derivadas.

*Obs: En Smalltalk todos los métodos son públicos y todos los atributos protegidos.*

*Mientras que en Java se puede utilizar cualquiera de las 3.*

*Puede ser:*

Nombre
-atributoUno:ClaseX
-atributoDos: ClaseY
+metodo():ClaseY
+metodo(parametro : ClaseZ)

- + Pública
- Privada
- # Protegida
- ~ Paquete

Pública	Privada	Protegida
El atributo o método puede ser accedido desde <u>afuera</u> del objeto.	El atributo o método solo puede ser accedido desde el mismo objeto. No es posible acceder tanto desde afuera como de las clases hijas.	El atributo o método solo puede ser accedido desde el mismo objeto <b>y</b> desde sus clases hijas.

## Inicialización y asociación

Si por cada objeto que creamos, inicializamos en cascada los objetos referenciados: quedarian atados al objeto de arranque de la asociación ⇒ esto se llama **composición**.

### Claves

- Hay varias maneras de delegar.
- Y al menos dos maneras de programar por diferencia.
- Herencia si se va a reutilizar la interfaz tal como está → relaciones “es un”
- Delegación cuando se va a reutilizar cambiando la interfaz.
- Redefinición permite cambiar implementación manteniendo la semántica.
- Clases abstractas no tienen instancias.

## Clase 4: Polimorfismo y Refactorización

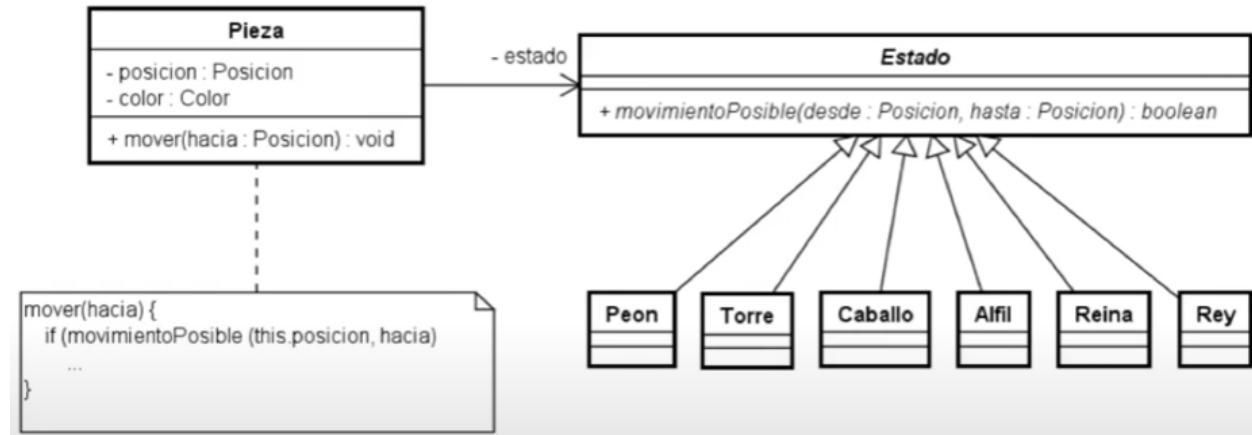
Definimos...

**Encapsulamiento:** Cada objeto es responsable de responder a los mensajes que recibe, sin que quien le envía el mensaje tenga que saber cómo lo hace.

**Polimorfismo:** Es la capacidad de respuesta que tienen distintos objetos de responder de maneras diferentes a un mismo mensaje.

# Soluciones Típicas

Se las llama “patrones”



Utilizamos el patrón State en el ejercicio del ajedrez. También existe el patron Template method, el Command y el Strategy.

## ¿Vinculación tardía?

Se plantea que el polimorfismo funciona porque la decisión del método a invocar se toma en tiempo de ejecución y no antes. Tiene sentido en lenguajes de comprobación estática.

# Interfaces

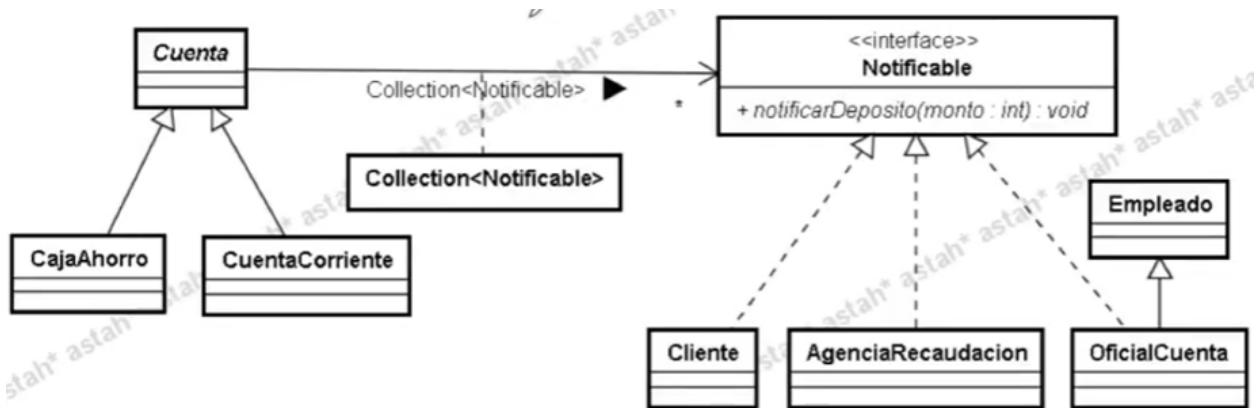
Son clases muy abstractas:

- Todos sus métodos son abstractos
- No posee atributos (sin estado)

Una Interfaz puede heredar de otra Interfaz.

Corolario: Si una clase declara implementar una interfaz y no implementa (redefine) uno de sus métodos es abstracta.

Ejemplo de interfaz en Java:



```

public class OficialCuenta extends Empleado
    implements Notifiable {
    ...
}
  
```

## Interfaces como protocolos

Son grupos de firmas de métodos:

- Sin implementar
- Indican maneras de comunicarse con los objetos

Una clase puede implementar varias interfaces

Entonces, ¿Qué es una interfaz?

▼ Visión del lenguaje

Una clase “muy abstracta” que se puede usar para herencia múltiple

▼ Visión desde el uso

Un tipo de dato que permite ver a un mismo objeto con distintos tipos

## Refactorizacion

### Entropía Creciente

Todo código va empeorando su calidad con el tiempo ⇒ entropía, degradación.

## Refactorización

Mejorar código, haciéndolo más comprensible sin cambiar funcionalidad.

¿Cómo? ⇒ Modificando su estructura interna y sin modificar el comportamiento observable.

ej: *Eliminar código duplicado, introducir polimorfismo.*

¿Para qué?

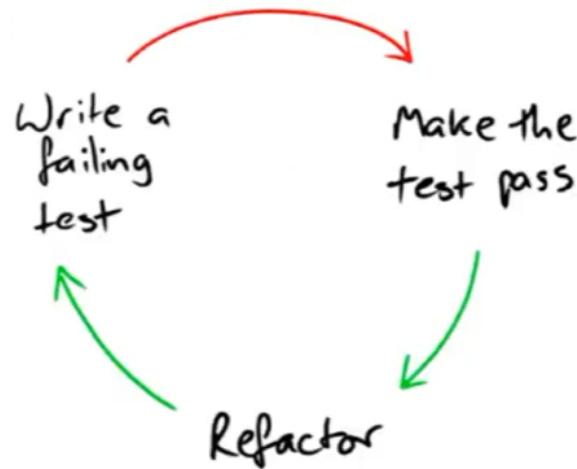
- Mejorar código, haciendolo más comprensible
  - Para modificaciones
  - Para depuraciones
  - Para optimizaciones
- Mantener alta la calidad del código (sino se degrada)
- A la larga, aumenta la productividad

¿Cuándo refactorizamos?

- Actitud constante
- Consecuencia de revisiones de código
- Antes de modificar código
- Después de incorporar funcionalidad
- Antes de optimizar (optimizar ≠refactorizar)
- Durante depuraciones

Problemas y refactorización → “**Bad smells in code**”, los llama Fowler

## Recapitulando TDD



## Claves

- Polimorfismo = distintos comportamientos para un mismo mensaje
- Polimorfismo seguro y sin herencia: Interfaces
- Manejar la entropía ⇒ Refactorización

# Clase 5: Excepciones y profundización de POO

## ¿Cuándo lanzar excepciones?

Si el contexto en el que estamos no hay suficiente información para resolver el portencial problema. Y la lanzamos a un contexto de nivel superior para que resuelva qué hacer.

En modo contractual, una excepción ocurre cuando no se cumple una precondición.

También para aislar el código que se une para tratar problemas del código básico (camino feliz).

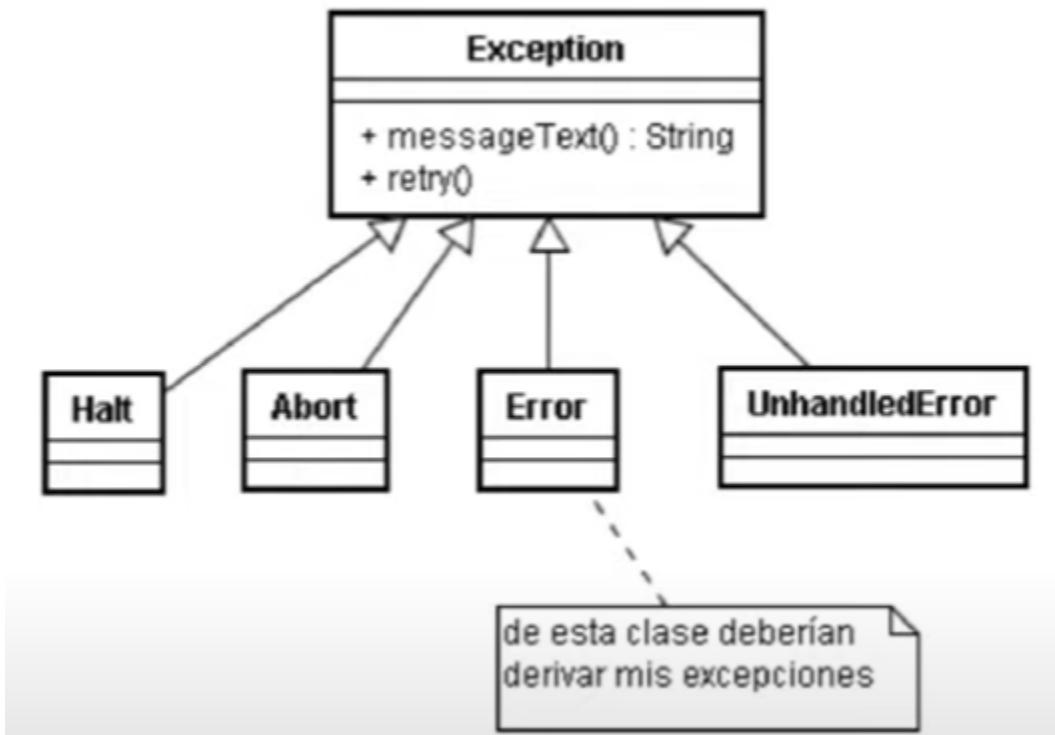
¿Qué hace al capturar?

- Resolverla mediante
  - Finalización súbita

- Continuación ignorando fallas
- Avance y recuperación
- Nuevo intento
- No resolverla y enviarla al contexto invocante
  - La misma
  - Otra excepción

## Jerarquía de Excepciones

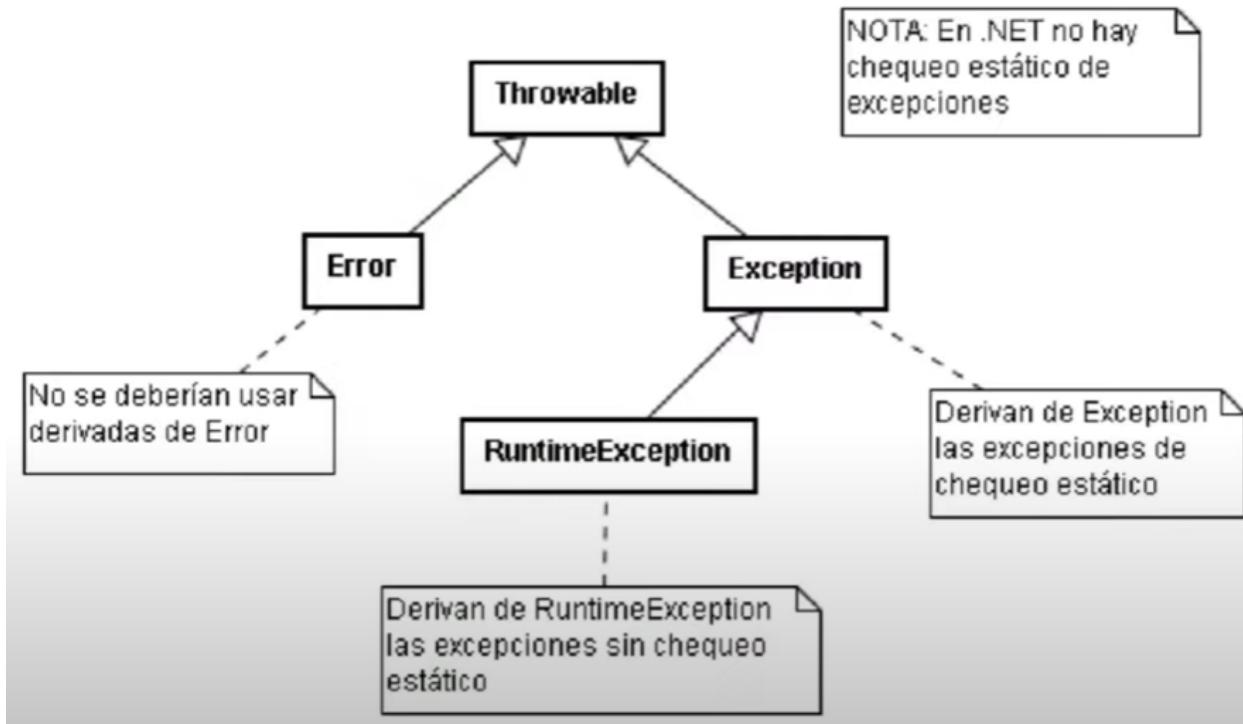
Cuando decidimos capturar un tipo de excepción, capturamos cualquier instancia de esa clase o una descendiente.



Sirven para dar mayor información sobre el tipo de problema.

Podrían agregar atributos y métodos, pero no es lo más habitual.

## Excepciones chequeadas en forma estática (Java)



Cláusula “throws” obligatoria.

En redefiniciones, mantener y no agregar → para mantener polimorfismo es muy molesto.

Obligación de capturar: chequeada por el compilador.

### Excepciones chequeadas

- son más seguras
- Molesta tener que capturarlas sí o sí
- Limita la redefinición, al no poder agregar nuevas excepciones → Se hizo cumplir el principio de sustitución.

*Obs: Microsoft diseñó .NET sin excepciones chequeadas.*

*Obs: Java permite ambas.*

## Unified Modeling Language (UML)

### Usos de UML

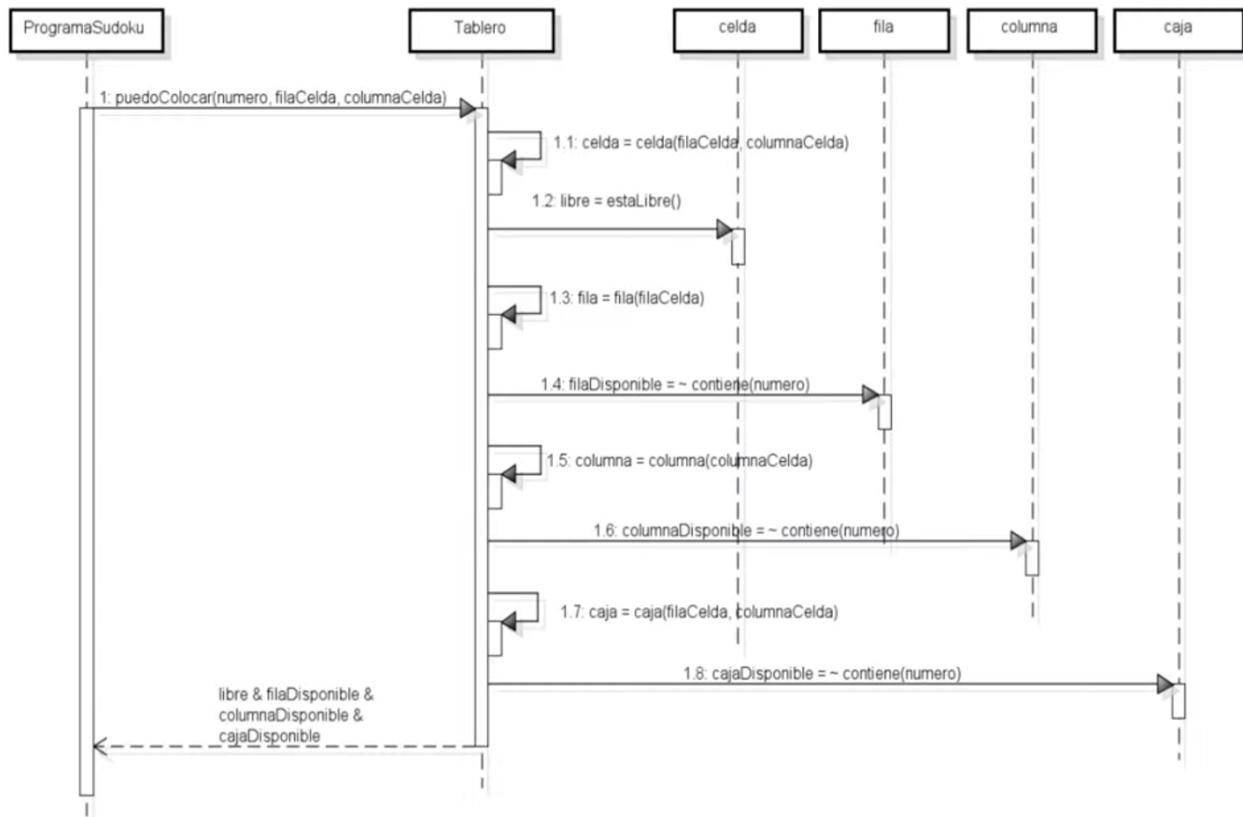
Para discutir diseños antes y durante la construcción.

Para generar documentos que sirvan después de la construcción.

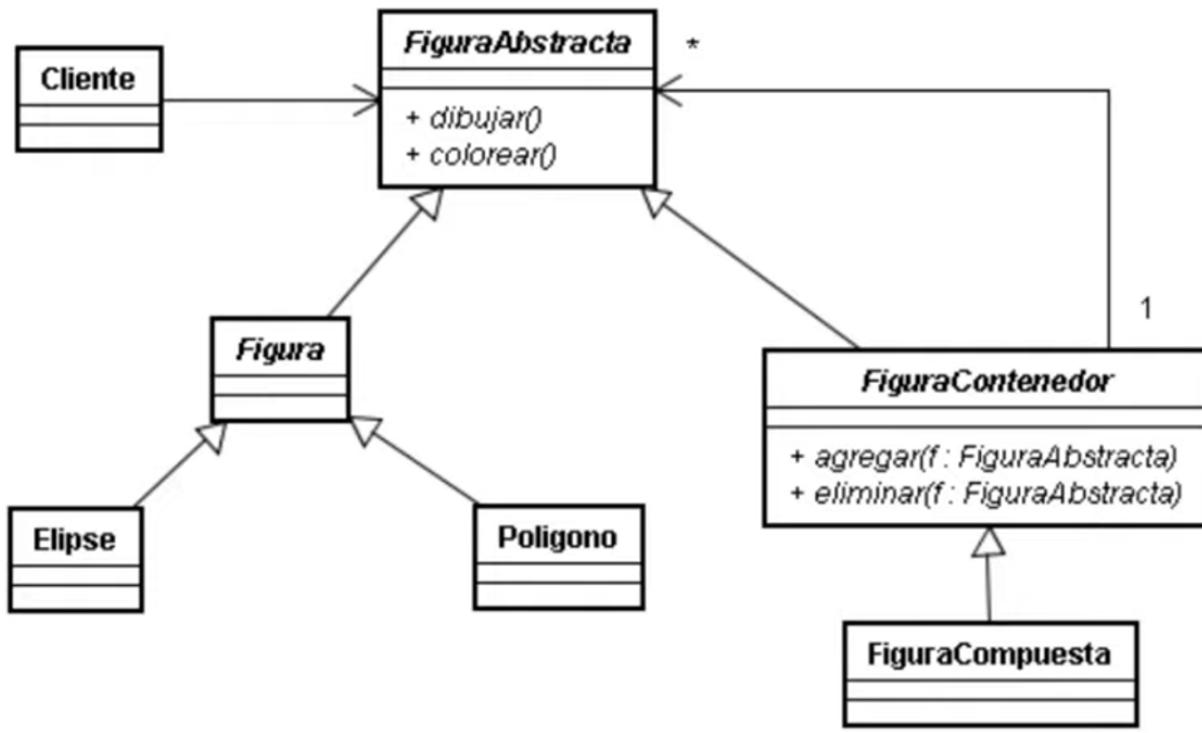
El destinatario es el ser humano.

Existen diagramas de Clase, Secuencia, Paquetes, Estado.

## Diagrama de Secuencia



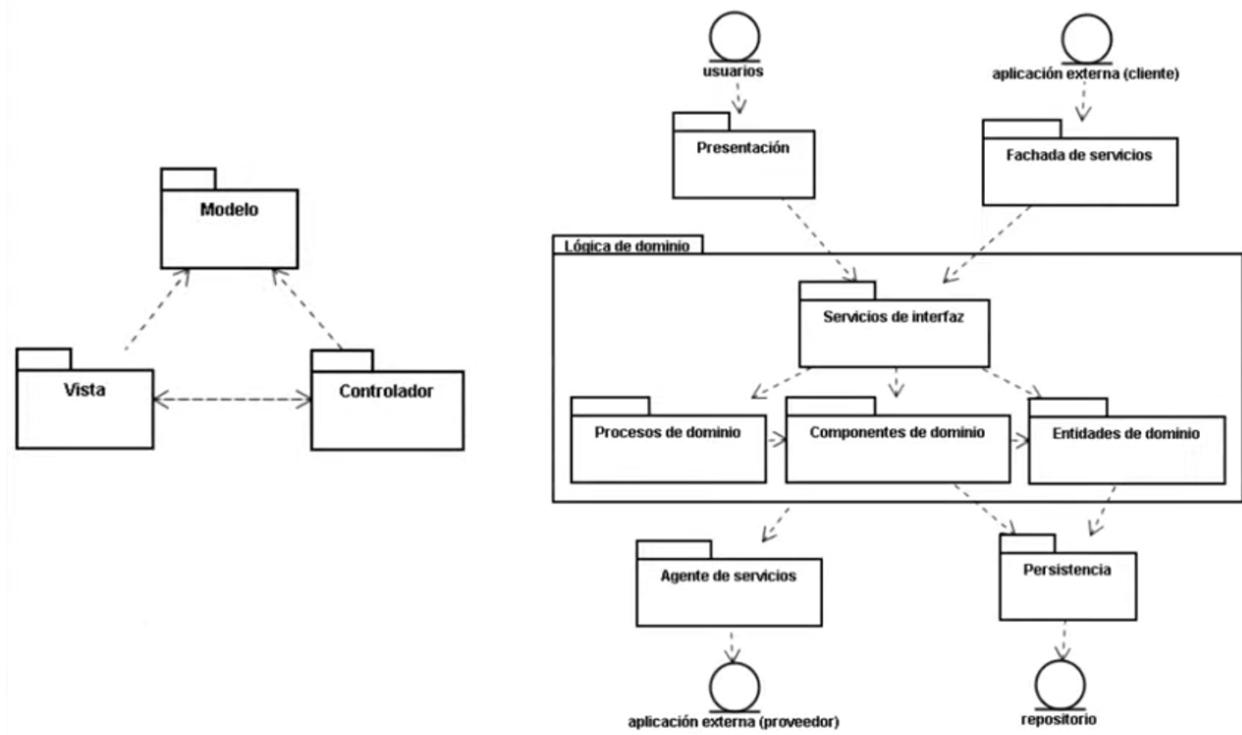
## Diagrama de Clases



## Diagrama de Paquetes

Agrupación de clases.

Para manejar la complejidad y modularizar.



*Obs: En Java, toda clase está en un paquete. Sirve para resolución de nombres. Cada clase pública en un archivo fuente separado.*

## Diagrama de Estado

### Estado

Representado por el conjunto de valores adoptados por los atributos de un objeto en un momento dado.

Situación de un objeto durante la cual satisface una condición, realiza una actividad o espera un evento.

### Evento

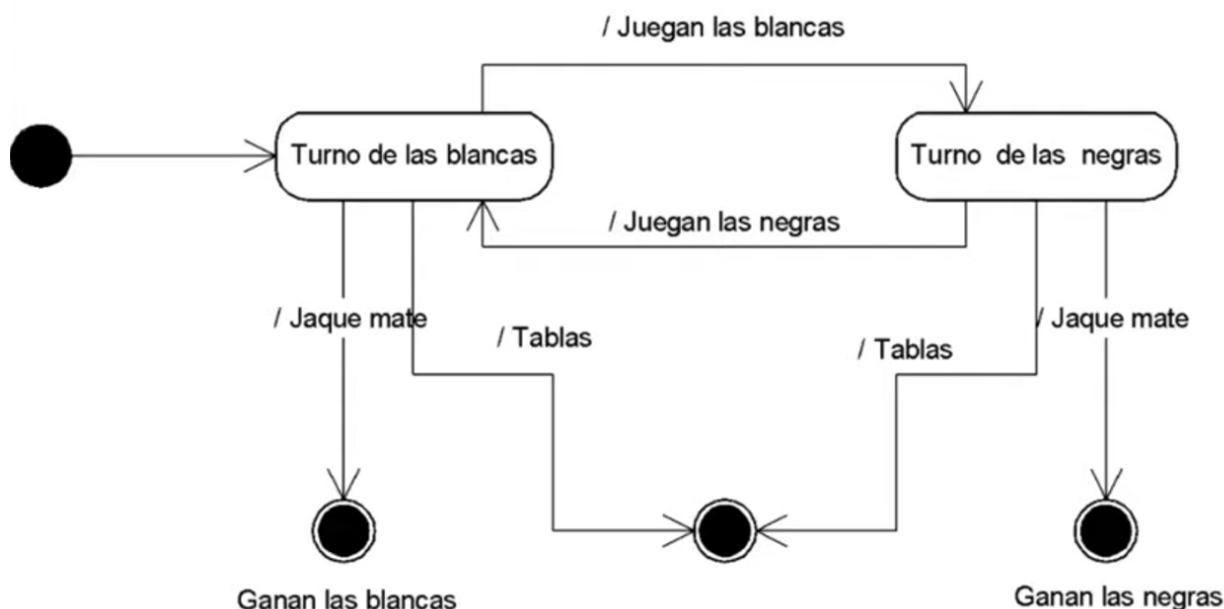
Estímulo que puede disparar una transición de estados.

Especificación de un acontecimiento significativo.

Señal recibida, cambio de estado o paso de tiempo.

Síncronico o asíncronico.

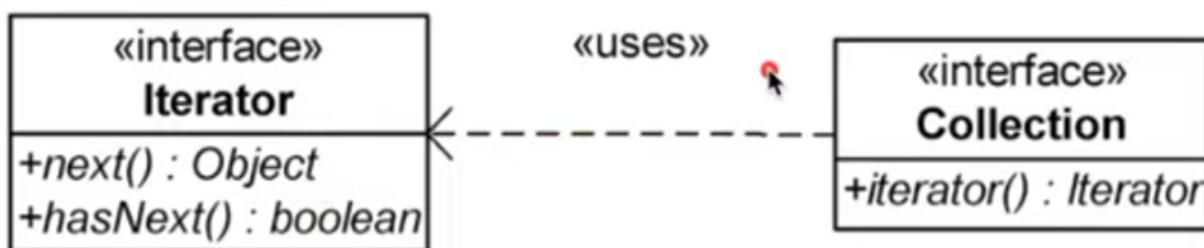
Ejemplo de ajedrez:



## Iterador

Son objetos que saben cómo recorrer una colección, sin ser parte de ella.

En Java, toda clase que implemente Collection puede generar un Iterador con el método iterator.



¿Para qué sirve?

- Llevan la abstracción a los recorridos de colecciones

- Facilitan cambios en la implementación
- No se necesita trabajar con el número de elementos
- Convierten a las colecciones en simples secuencias

## Claves

Inicialización debería dejar al objeto en un estado válido

Excepciones cuando no hay suficiente información en el contexto del problema

UML es una herramienta de modelado

- Para discutir diseños antes del código
- Para generar documentos que sirvan después de la construcción

Genericidad lleva la idea del tipo estático a las colecciones

# Clase 6: Cálidad de Código; XP

*“Los necios obedecen las reglas; los sensatos, las usan como guía”* (Douglas Bader).

*“El código es el único artefacto del desarrollo del software que siempre se va a construir”*(Carlos Fontenla).

*“I'm not a great programmer; I'm just a good programmer with great habits”*(Kent Beck).

## Legibilidad

El código se escribe una vez y se lee muchas.

Principios de Legibilidad de Código:

- No repetir: extrayendo métodos o clases(por delegación o generalización).
- Estandarizar: nombres, formatos, comentarios, etc.
- No Sorprender: Suponer igual nivel del desarrollador que va a necesitar mi código.  
Evitar lucirse. No explicar lo obvio.

Nombres:

- De variables:
  - Que sean descriptivos.
  - Términos del problema, no de la solución.
  - Evitar significados ocultos
  - Variables booleanas
- De métodos:
  - Que describa todo lo que hace el método
  - Que describa la intención (el qué) y no detalles de implementación (el cómo)

## Métodos

Tratar de que no dependan de un orden de ejecución.

Recursividad sólo cuando ayuda a la legibilidad.

Que afecte a la clase en la que está declarado.

### Parámetros en los métodos:

- Que no sean muchos.
- Que estén ordenados con algún criterio.
- Chequear valores válidos a la entrada.
- Los valores de entrada no deben cambiarse.

### Inicialización de variables:

- Siempre explícita.
- Un uso para cada variable.

- Vida de la variable lo más corta posible.
- Usar métodos booleanos para guardar partes de tests complicados.
- Evitar **números magicos** (números hardcodeados).

## Comentarios

- Son buenos para aclarar código o como resumen de una secuencia de acciones.
- No repetir en los comentarios lo que dice el código.
- Tienen que ser fáciles de mantener.
- El comentario debe ir antes del código al que se refieren.
- Casos especiales a documentar con comentarios.

## Mejora de Desempeño

Resistir la tentación: No siempre el usuario percibe las mejoras en el código, sino que privilegia su propia experiencia.

Cuando todo falle, mejoramos el código pero solo en las partes críticas.



### Algunas Posibilidades

- Evitar el uso de memoria externa cuando se puede usar memoria interna.
- En las evaluaciones lógicas, utilizar la opción de cortocircuito.

*En un “(A & B) ifTrue:” no necesitamos evaluar b si a es falso.*

- Ordenar las preguntas en acciones if compuestas y switch: Conviene evaluar primero las más frecuentes.
- Calcular todo lo que se pueda antes de entrar a un ciclo.
- Usar tipos enteros en vez de punto flotante.
- Tabular datos en vez de usar funciones trascendentes.
- Liberar memoria que ya no se necesita, si el entorno no lo hace solo.
- Usar lenguajes compilados o de menor nivel.

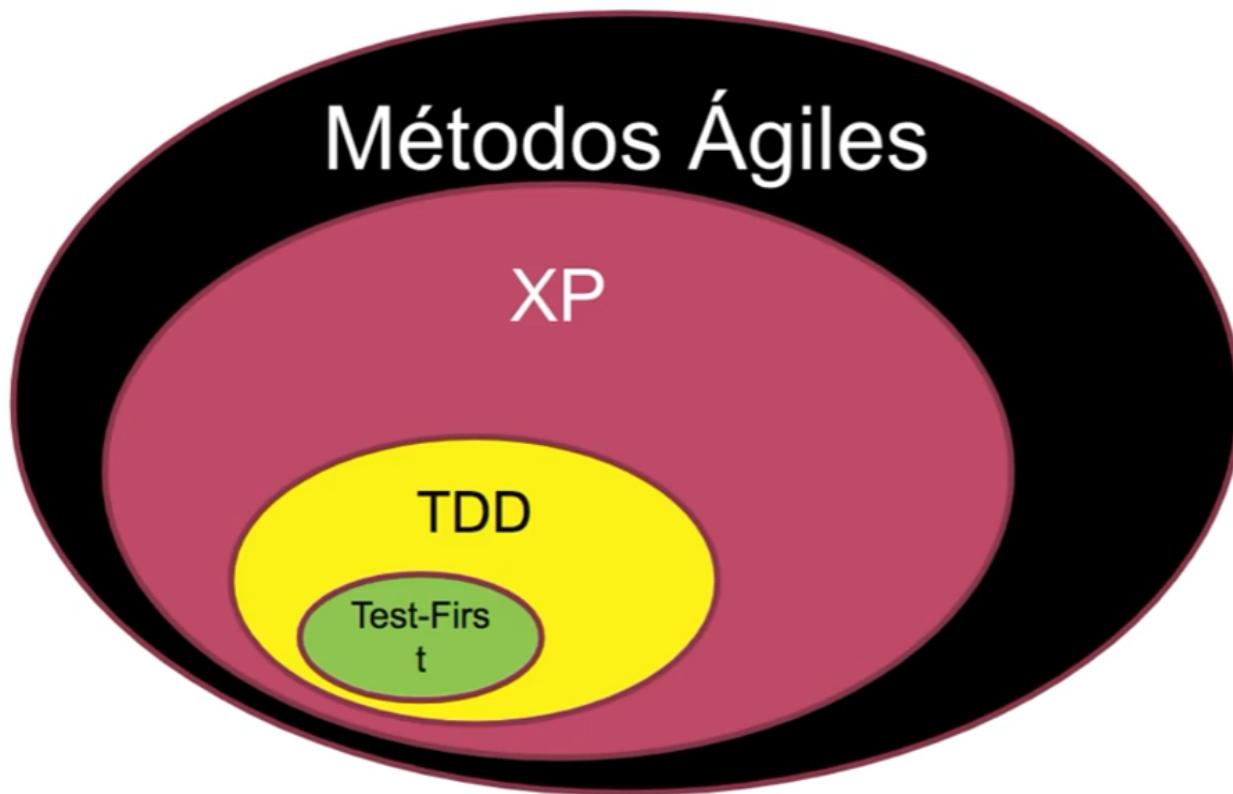
### Dichos Importantes

*“Premature optimization is the root of all evil” (Donald Knuth)*

“1. Make it work. 2. Make it right. 3. Make it fast.” (Kent Beck)

## Extreme Programming (XP)

Es una metodología incremental formulada por Kent Beck



Es un conjunto de prácticas de desarrollo centradas en la programación.

Es uno de los “métodos ágiles”

“Llevan al extremo las buenas prácticas de programación y cuestiones de sentido común”

### Buenas prácticas XP

- Simplicidad: *“the simplest thing that could possibly work”*
- Propiedades

- Código autodocumentado
- Embebida en el código (comentarios)
- Pruebas unitarias
- Otras pruebas automatizadas
- Menor valoración a
  - UML
  - Documentos externos
- TDD
- Integración continua
- Refactoring
- **Pair Programming:** requiere que dos programadores participen en un esfuerzo combinado de desarrollo en un sitio de trabajo. Mientras que uno codifica las pruebas de unidades el otro piensa en la clase que satisfará la prueba.

## Clase 7: Diseño Orientado a Objetos

### Diseño

Concebir una solución para un problema. Es el “cómo” del desarrollo y entre el “qué” de las necesidades del usuario y la propia construcción del producto.

#### ¿Para qué diseñamos?

El software debe ser flexible:

- Permitir el cambio
- Adaptarse a nuevas tecnologías
- Adaptarse a nuevos dominios

Principio N1: Prepararse para el cambio

# **Modularización**

## **Cohesión y acoplamiento**

Para mantener comprensibles, mantenibles y reutilizables a los módulos

- Módulos de alta cohesión
  - Se refieren a la coherencia intrínseca del módulo
  - Visión interna y externa
  - Pero respecto del módulo solamente
- Módulos de bajo acoplamiento
  - Se refiere a la relación con otros módulos
  - Vision externa
  - Mira más bien al sistema

## **Acoplamiento de Clases**

- La clase representa una sola entidad (un sustantivo debe poder describirla)
  - Para lograr alta cohesión
  - Integridad conceptual
- La clase debe tener pocas dependencias con otras clases
  - Para lograr bajo acoplamiento
  - Modificaciones tienen efectos acotados

## **Niveles de dependencias entre clase (mayor a menor)**

## Herencia (de clases o interfaces)



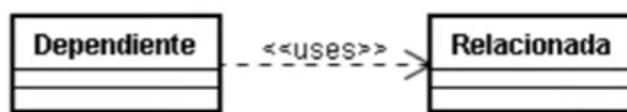
## Composición



## Asociación simple



## Dependencia débil



### Cohesión y acoplamiento en paquetes

- Los paquetes deben tener pocas dependencias con otros paquetes
  - Para lograr bajo acoplamiento
  - Modificaciones tienen efectos acotados
- Los paquetes deben describir una visión global del sistema
  - Con clases relacionadas entre sí, para lograr alta cohesión

### Cohesión y acoplamiento en métodos

- Cada método debe poder describirse con una oración que tenga un solo verbo activo
  - Para lograr alta cohesión
  - Mejora la legibilidad de todo el código

- El método debe tener pocos parámetros y pocas llamadas a otros métodos
  - Para lograr bajo acoplamiento
  - Modificaciones tienen efectos acotados
  - Parámetros por valor en lo posible

## Diseño de Clases

Razones para crear clases:

- Modelar una entidad del dominio.
- Modelar una entidad abstracta.
  - Al encontrar elementos en común entre varias clases.
  - Clase abstracta o interfaz, cuando nos interese mantener la interfaz.
  - Clase delgada, cuando nos interese los datos en común.
- Reducir complejidad: una sola abstracción por clase.
- Aislard complejidad y detalles de implementación en una clase aparte.
- Aislard una porción de código que se espera que cambie respecto de otras más estables.
- Al separar código poco claro en clases especiales.
- Agrupar operaciones relacionadas en una clase de servicios.

Cuando NO crear una clase:

- Una nueva clase descendiente debe añadir o redefinir un método (modificar la interfaz), sino no es necesaria

## Principios SOLID (Robert Martin)

- S ⇒ Single Responsibility (SRP) → Única Responsabilidad

Toda clase debe tener una única responsabilidad. Separar por uso o razón de cambio. tiene que ver con la alta cohesión.

- O ⇒ Open-Closed (OPC)

Las clases tienen que estar cerradas para modificación, pero abiertas para reutilización. No modificar las clases existentes, sino extenderlas o adaptarlas por herencia o delegación.

- L ⇒ Liskov Substitution (LSP) → Sustitución

Las subclases deben poder ser utilizadas a través de la clase madre sin ningún problema (sin saber la diferencia). En los clientes, debemos poder sustituir un tipo por sus subtipos: relación “es un”. La subclase sea un subconjunto de la clase.

Las clases base no deben tener comportamientos que dependan de las clases derivadas.

- I ⇒ Interface Segregation (ISP)

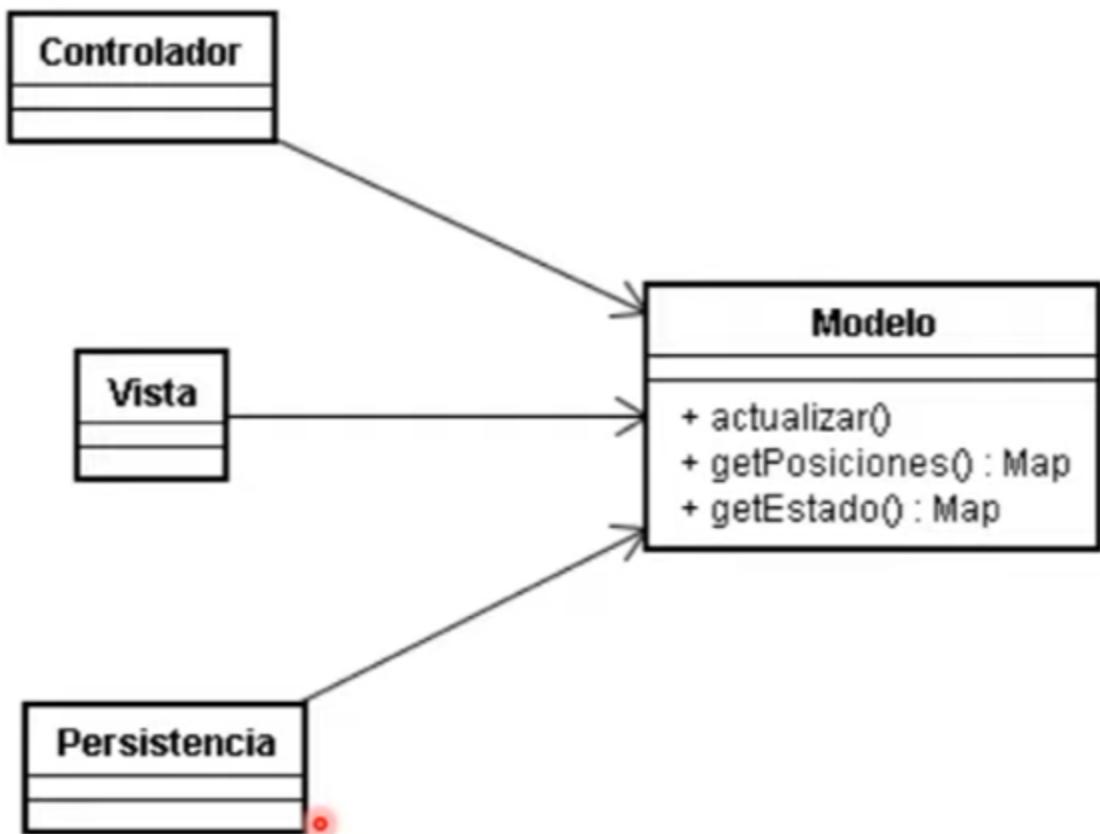
Los clientes de una clase no dependan de los métodos que no utilizan (Los clientes no deben ser forzados a depender de métodos que no utilizan). Se apoya en otros principios. Muchas interfaces específicas son mejor que una general.

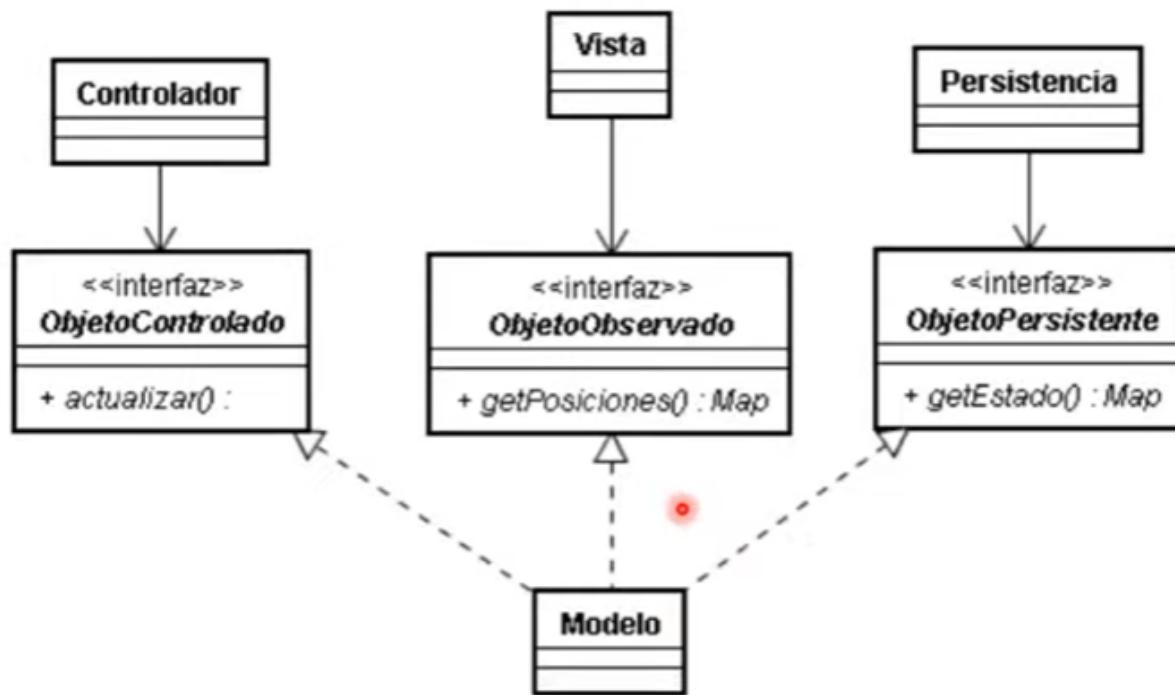
- D ⇒ Dependency Inversion (DIP)

No depender de clases concretas volátiles: no conviene que una clase herede o tenga una asociación hacia una clase concreta que tiene alta posibilidad de cambio. Se debe depender de las abstracciones y no de las implementaciones.

Las clases abstractas y las interfaces son más estables: conviene utilizarlas para herencia o delegaciones.

## Segregación de Interfaz





Modelo implementa tres interfaces, y cada tipo de objeto la utiliza solamente a través de cierta interfaz

## Delegación sobre herencia

La Herencia es estática mientras que la Delegación otorga mayor flexibilidad.

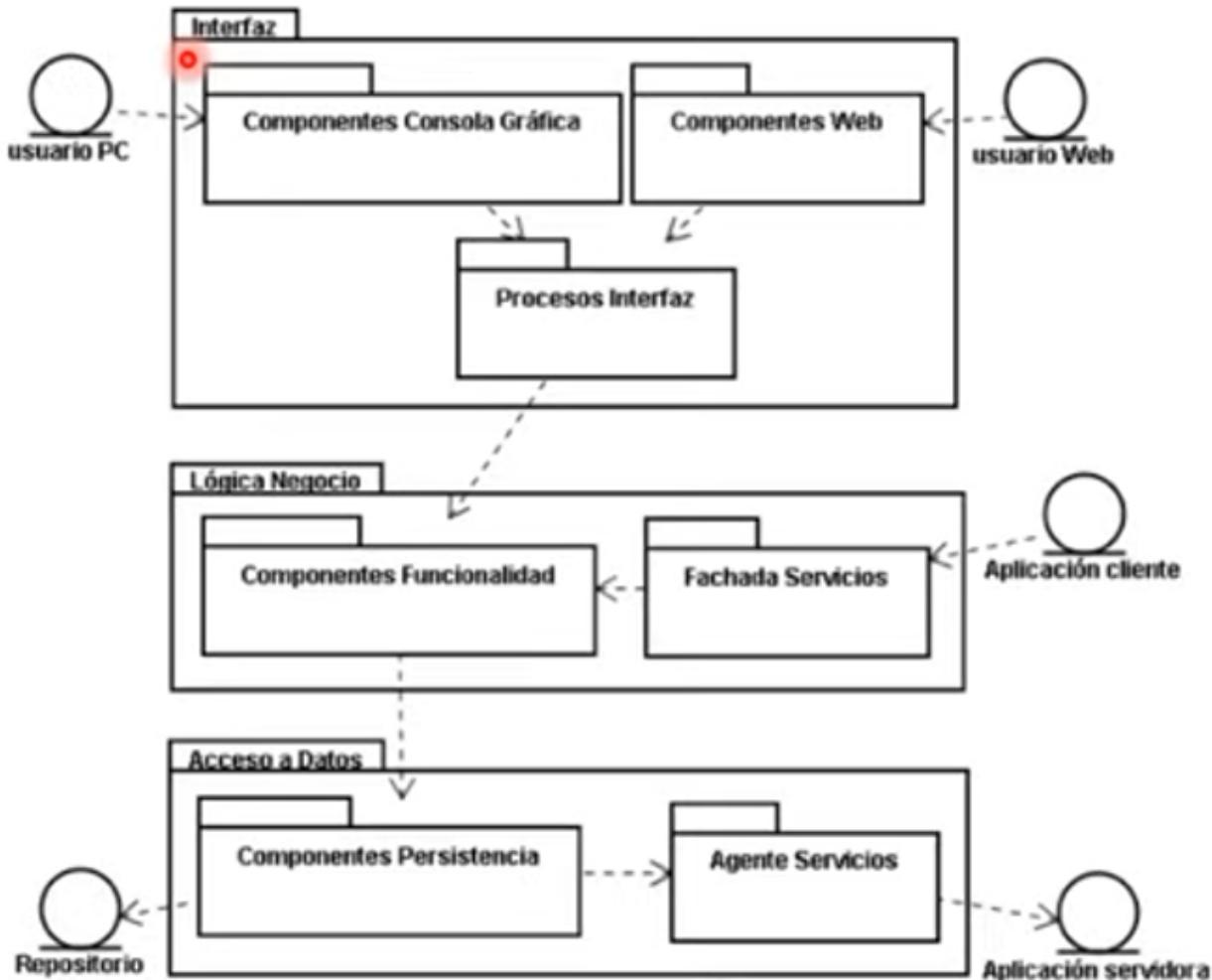
### ¿Cuándo usar?

Herencia (concreta, abstracta o de interfaces) cuando se va a reutilizar la interfaz.

Delegación cuando se va a reutilizar algunas responsabilidades (comportamiento o estado).

## Diseño de Paquetes

Permiten un nivel de abstracción más alto. Deben describir una situación más global de un sistema



- Bajo Acoplamiento: usar diagrama de paquetes como fuía para ver dependencias.
- Construir las agrupaciones de clases de modo tal que sirvan para reutilizarse.
- Las clases que se prevé que se van a usar conjuntamente deberían colocarse en el mismo paquete.
- Las clases que cambian un poco deberían ubicarse en paquetes separados de aquellas que cambian mucho.

## ¿Cuándo crear métodos?

Situaciones típicas de refactorización:

- Reducir complejidad

- Hacer legible un pedazo de código
- Simplificar tests booleanos
- Evitar duplicación de código
- Encapsular complejidad y dependencias de plataforma
- Mejorar cohesión

Siempre que genero métodos innecesarios para el cliente los hago privados

Cada método debe tener un propósito simple

## Visibilidad

Todo debe ser lo más privado que se pueda: principio de mínimo privilegio aplicado a clientes.

Garantiza que se pueda modificar código afectando al mínimo posible de clientes.

Atributos privados.

Métodos según lo que se requiera (no siempre públicos).

Propiedades, en general, públicas (pero el "get" o el "set" puede que no).

# Clase 8: Metodologías de Desarrollo de Software

## Desarrollo de Software

Desarrollo ≠ Programación → Desarrollo de software incluye a la Programación



### **Disciplinas del desarrollo (operativas)**

- Captura de requisitos: qué necesita el cliente
- Análisis: qué vamos a construir
- Diseño: cómo
- Construcción o implementación
- Pruebas: verificación y validación
- Despliegue (en hardware)

### **Disciplinas del desarrollo (soporte)**

- Administración del proyecto, incluyendo seguimiento y control de tiempos y costos
- Gestión de cambios
- Administración de la configuración
- Gestión de los recursos humanos y la comunicación

- Gestión de riesgos
- Gestión de proceso

## Características del Software

- Intangible.
- Maleable: posibilidad de cambio.
- Se desarrolla por proyectos o se mantiene en el tiempo como producto.
- Alto contenido intelectual.
- El software no se “fabrica”, sino que se construye o se desarrolla.
- Mantenimiento constante.

## Fracasos del desarrollo de software

- Proyectos que no terminan a tiempo.
- Proyectos que cuestan más que lo estimado.
- Accidentes.
- productos que no cumplen lo que el solicitante quiere.

## Algunas reflexiones

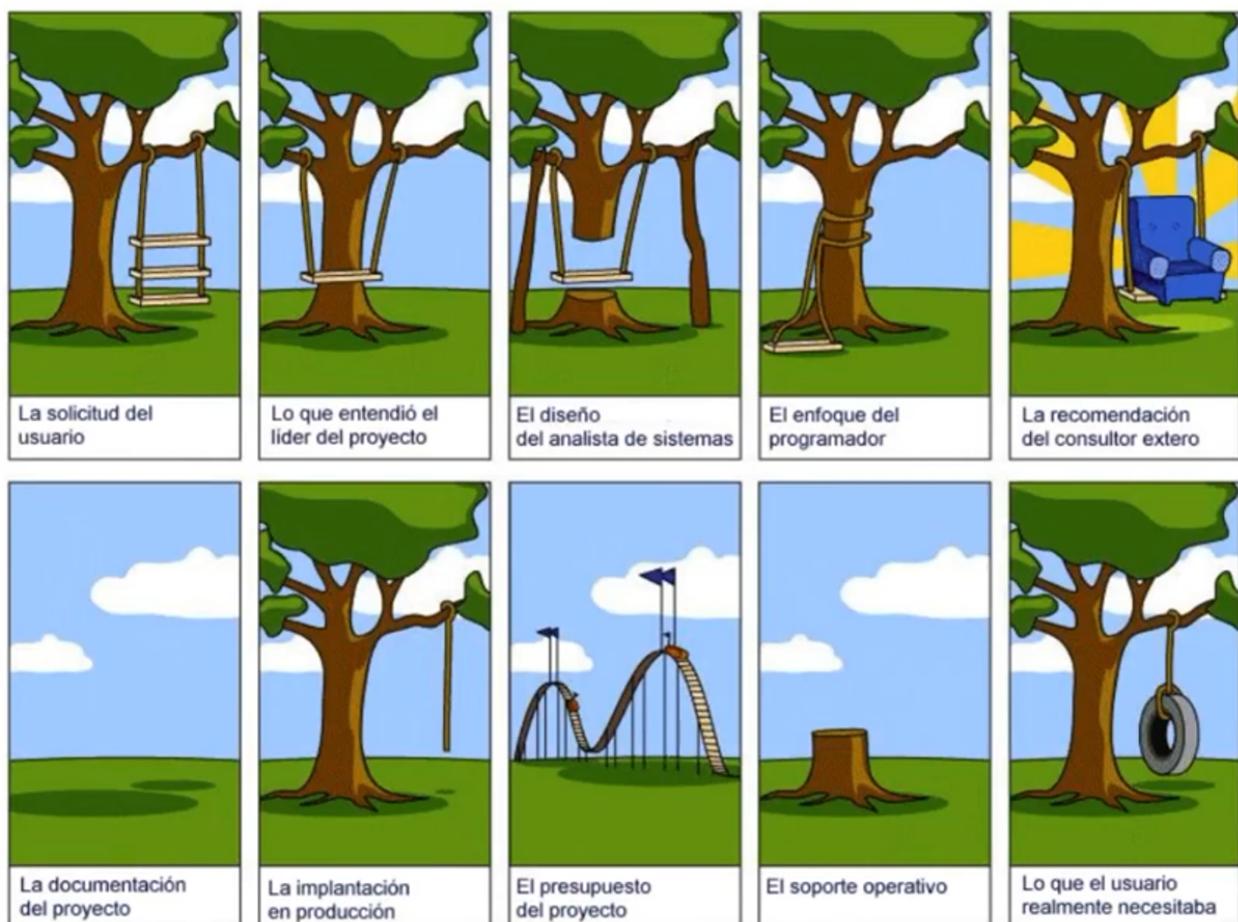
- Los problemas del desarrollo no son sólo tecnológicos.
- **Ley de Brooks:** agregar gente a un proyecto atrasado lo atrasa más.
- Cuidar la comunicación.

## Metodología

Método o proceso

- Define quién debe hacer qué, cuándo y cómo se deben realizar las distintas tareas.
- proceso unificado, Extreme Programming, Scrum, yourdon, etc.
- Sirve para estructurar, planificar, desarrollar y controlar el desarrollo de software.

- Determina
  - Fases
  - Roles
  - Actividades
  - Artefactos
  - Etc.



## Ciclos de vida de desarrollo de software

### Lineal

Proyecto ⇒ serie de pasos que implican actividades distintas

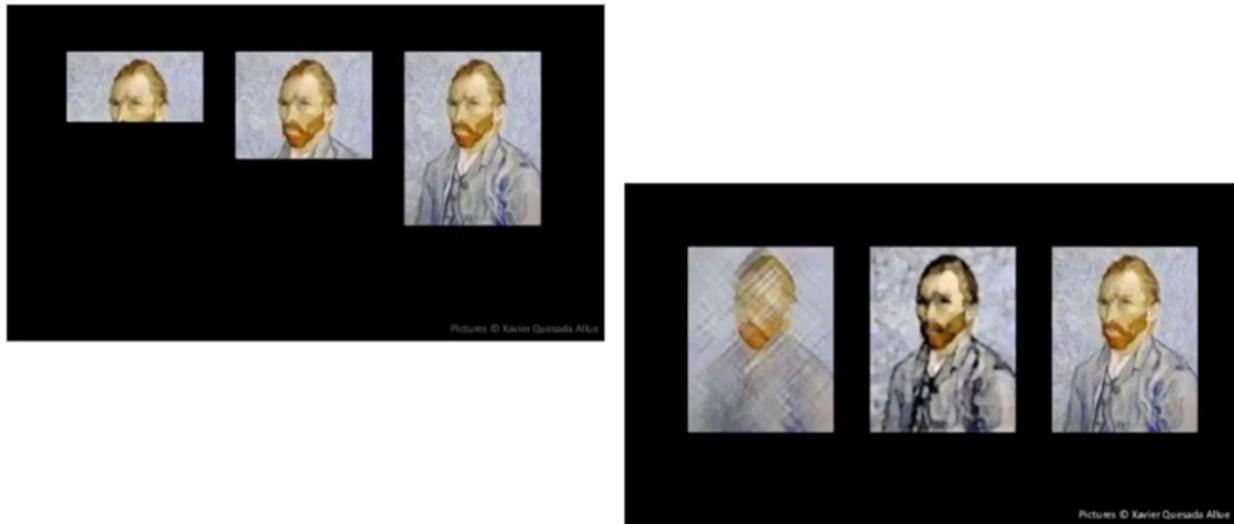
### Método Cascada



## Incremental

Se llega a una solución cada vez más refinada, o que incluya más características, o ambas.

Se puede partir de una visión aproximada. Los argumentos no se conocen de entrada...

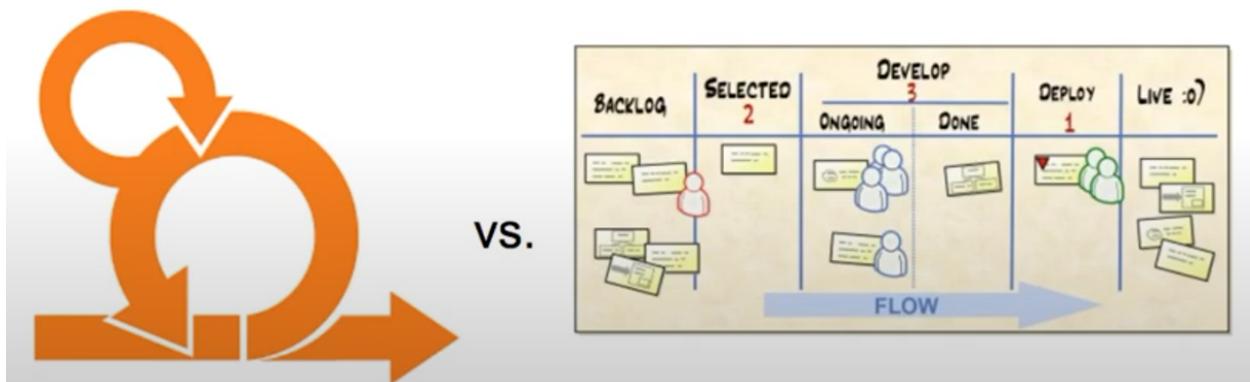


**¿Por qué conviene incremental?** El software evoluciona mediante la interacción durante su construcción.

**Sí se discute..**

¿Iterativo?: por interacciones con comienzo y fin

¿Continuo?: desarrollo a base de funcionalidades sin hitos definidos



## Categorías de Métodos

1. Procesos predictivos
  - Planificación más detallada y rígida.

- Pensados para grandes proyectos: inaceptablemente pesados para sistemas pequeños o medianos.
- Destacan el Proceso Unificado (UP), TSP, Cleanroom

## 2. Métodos ágiles o adaptables

- Más abiertos a los cambios.
- Permiten organizar desarrollos sin caer en burocracia inútil.
- Nacieron como alternativa a carecer de metodología.

## Manifiesto Ágil

Más importancia	sobre
Individuos e interacciones	Procesos y herramientas
Software funcionando	Documentación extensiva
Colaboración con el cliente	Negociación contractual
Respuesta ante el cambio	Seguir el plan

### Características

- Maleable ⇒ no impedir los cambios
- Particionable ⇒ desarrollar en forma incremental
- Extensible ⇒ pensar en evolución permanente
- Software es materialización del conocimiento ⇒ interacción colaborativa

### Equipos

- Los que desarrollan son personas ⇒ potenciar capacidades sociales.
- Las personas deben comunicarse efectivamente ⇒ reuniones cara a cara y abiertas.
- Tendemos a querer ser parte de equipos exitosos ⇒ autoorganización.
- Únicos animales que podemos reflexionar sobre nuestros errores ⇒ retropectivas frecuentes.

## Algunos Métodos Ágiles

1. Extreme Programming (XP)
  - de Kent Beck y la comunidad Smalltalk.
  - Lleva al extremo las buenas prácticas.
2. Scrum
  - de Ken Schwaber y Mike Beedle.
  - Provee roles y artefactos centrados en seguimiento y control del proyecto.

### Claves

- Problemas desarrollo >> Problemas tecnológicos
- Desarrollo ⇒ construcción de conocimiento
- Naturalmente, el software se construye de manera incremental
- Los métodos ágiles se adaptan mejor a las características del software
- Elegir y adaptar los métodos de desarrollo → esa es la tarea de un ingeniero

## Clase 9: RTTI y reflexión

### RTTI

Información de tipos en tiempo de ejecución

Nos permite conocer la clase exacta de un objeto o a la familia de la clase del objeto.

#### Conocer la clase

## Java

```
assertTrue (cajaAhorro.getClass( ) == Cuenta.class);
assertTrue (ctaCte.getClass ( ) ==
CuentaCorriente.class);
assertFalse (ctaCte.getClass ( ) == Cuenta.class);
```

## Smalltalk

```
self assert: (cajaAhorro class = Cuenta).
self assert: (ctaCte class = CuentaCorriente).
self deny: (ctaCte class = Cuenta).
```

## Conocer la familia

## Java

```
assertTrue (cajaAhorro instanceof Cuenta);
assertTrue (ctaCte instanceof Cuenta);
assertTrue (ctaCte instanceof CuentaCorriente);
```

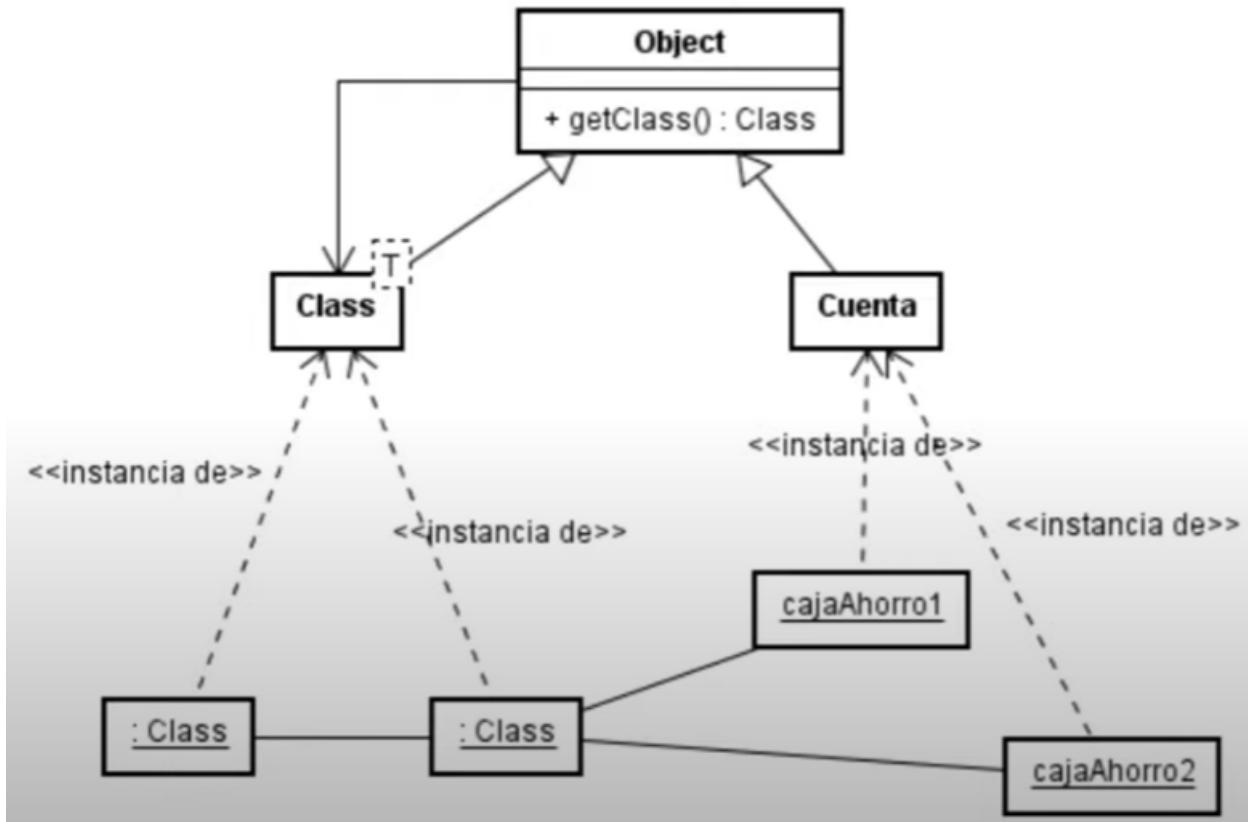
## Smalltalk

```
self assert: (cajaAhorro isKindOf: Cuenta).
self assert: (ctaCte isKindOf: CuentaCorriente).
self assert: (ctaCte isKindOf: Cuenta).
```

## Problemas con RTTI

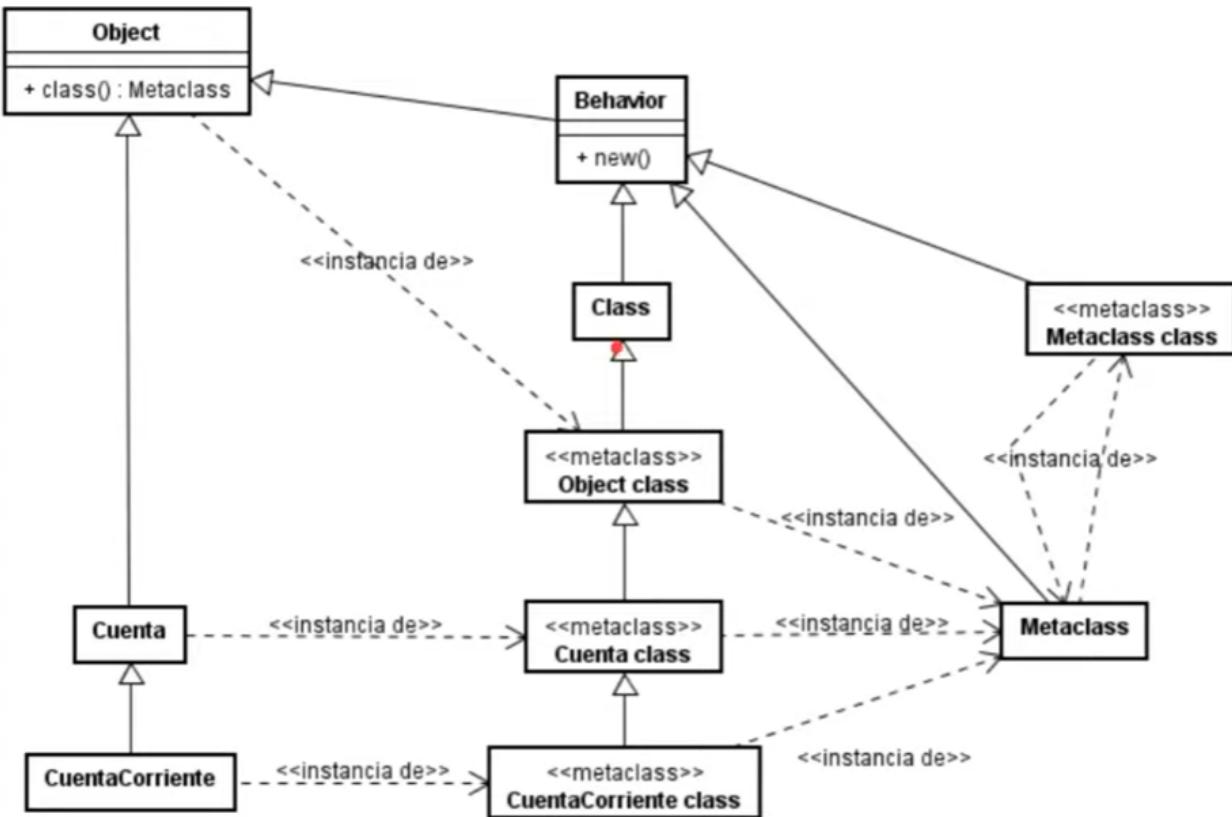
- Compromete la extensibilidad: es la anti-programación orientada a objetos
- Evita el polimorfismo

## RTTI en Java



## RTTI en Smalltalk

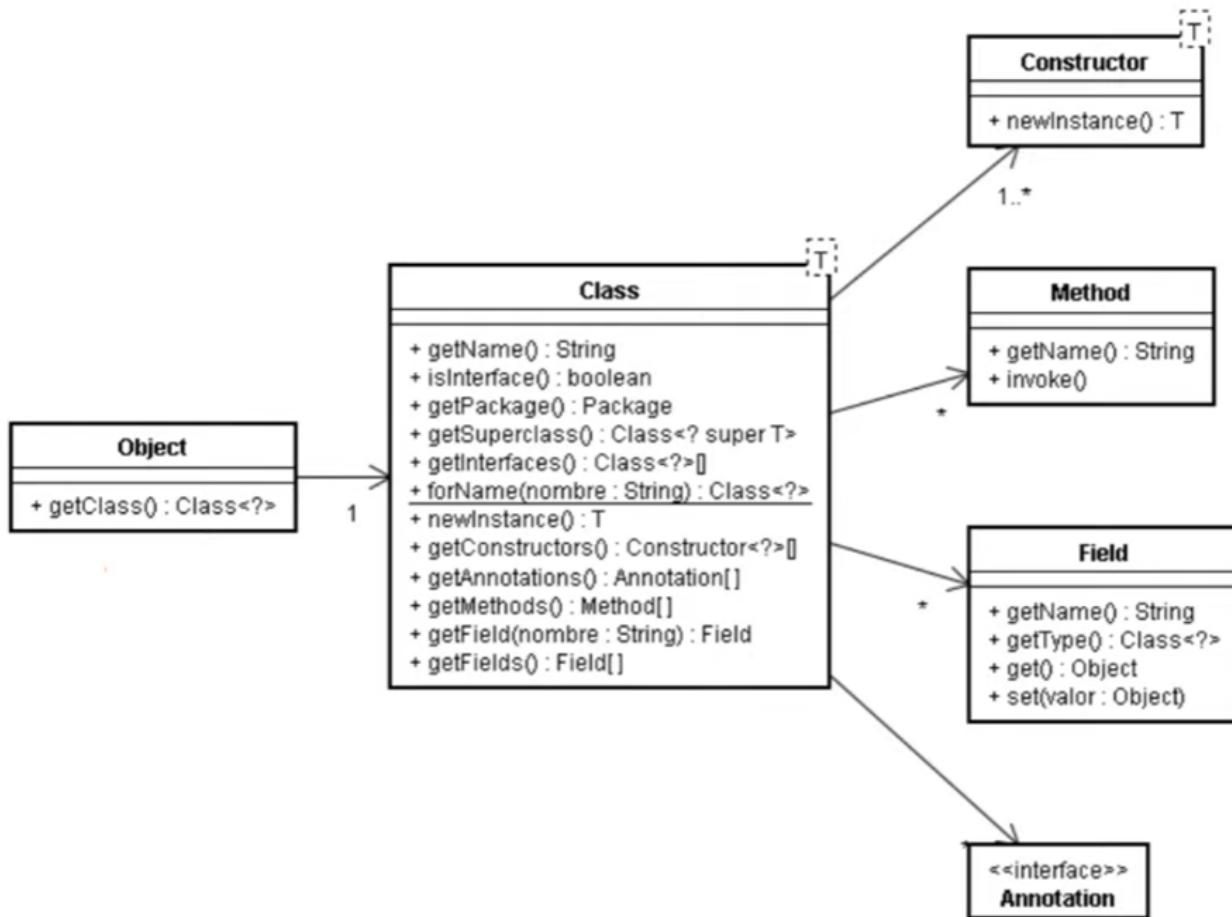
- Las clases son objetos
- Las clases son instancias de una metaclas
- Hay una jerarquía de metaclasses paralela a la de la clase
- Cada metaclas hereda de Class
- Class hereda de Behavior
- Las metaclasses son instancias Metaclass



## Reflexión

Lo que me permite es interactuar con los distintos objetos en tiempo de ejecución a través de RTTI.

**En Java:**



RTTI: El compilador debe conocer los tipos.

¿Qué pasa si recibo un objeto del que no sé nada? Puedo obtener el objeto Class y preguntar por métodos, atributos, interfaces, etc.

La información debe estar en tiempo de ejecución.

Reflexión ya lo utilizamos en el Framework JUnit y SUnit (como en métodos “public void testXxx()”). Así mismo utiliza polimorfismo (como en métodos setUp() y tearDown()). Esto es algo bastante común en todos los frameworks.

## En Smalltalk:

Es más potente que en Java. posee 2 niveles:

- Introspección → Similar a Java
- Intersección

- Actuación sobre el entorno de ejecución
- Admite la metaprogramación

## Inrospección

- Hay más control sobre el entorno de ejecución que en Java
- Todos son objetos (Workspace, Debugger, Inspector, Transcript)
- Hay más información
  - Subclases
  - Instancias existentes
  - Toda la jerarquía de herencia
  - Uso de objetos desde métodos
  - Referencias cruzadas entre métodos

## Intersección y metaprogramación

- Control total de los objetos durante la ejecución
- Permite la metaprogramación
- Cambios de comportamiento en forma dinámica

*Obs: Ojo con reflexión: podemos terminar generando cualquier cosa → difícil de testear, difícil de leer y no cualquiera lo usa bien.*

# Java

## Objetivos

1. Disminuir la dependencia de plataformas específicas
  - a. Sin recompilación
  - b. Plataforma = hardware + sistema operativo

2. Sin grandes pretensiones de tiempo real y control explícito de memoria
3. Foco en bajo costo de desarrollo

### Lenguajes creados para JVM

Java  
BBj  
Clojure  
Fantom  
Groovy  
MIDletPascal  
Scala  
Kawa

### Adaptaciones de otros

Erjang (Erlang)  
Rhino (JavaScript)  
Free Pascal (Pascal)  
Quercus (PHP)  
Jython (Python)  
NetRexx (REXX)  
JRuby (Ruby)  
Jacl (TCL)



## Smalltalk

Ecosistema innovador

- Interfaces GUI / WIMP
- IDE integrado
- Herramientas de refactorización integradas
- Multimedia e hipertextos
- Prototipado rápido de aplicaciones

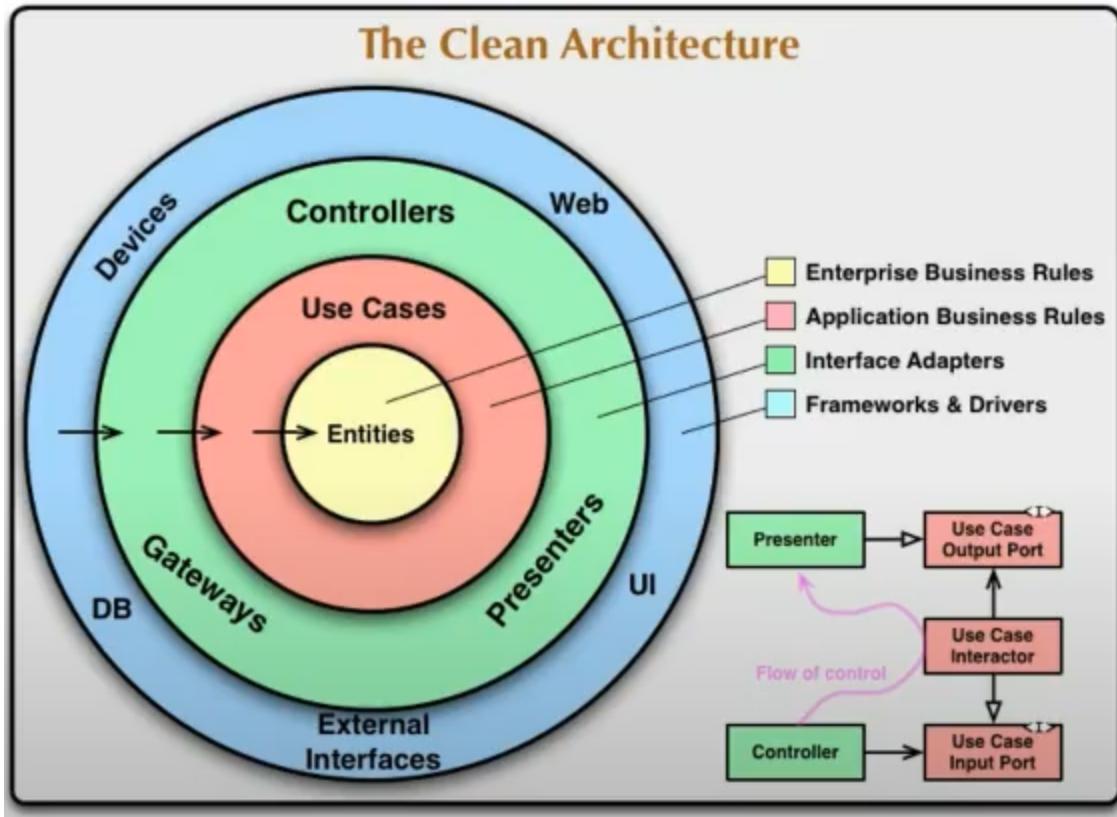
- Persistencia resuelta para la imagen
- Primeros patrones de diseño (MVC especialmente)
- Métodos ágiles (especialmente XP)

### Claves

- Suele ser posible saber la clase de una instancia en tiempo de ejecución
- En Smalltalk todo es un objeto
- Smalltalk y Java surgieron de ecosistemas innovadores

## Clase Complementaria: Presentación MVC

**Mecanismo de despacho** → como disponibilizamos la app, API, etc (ej: github pages)



*Obs: Nosotros trabajamos con Entities y Use Casas.*

¿Qué busca?

Disponibilizar software

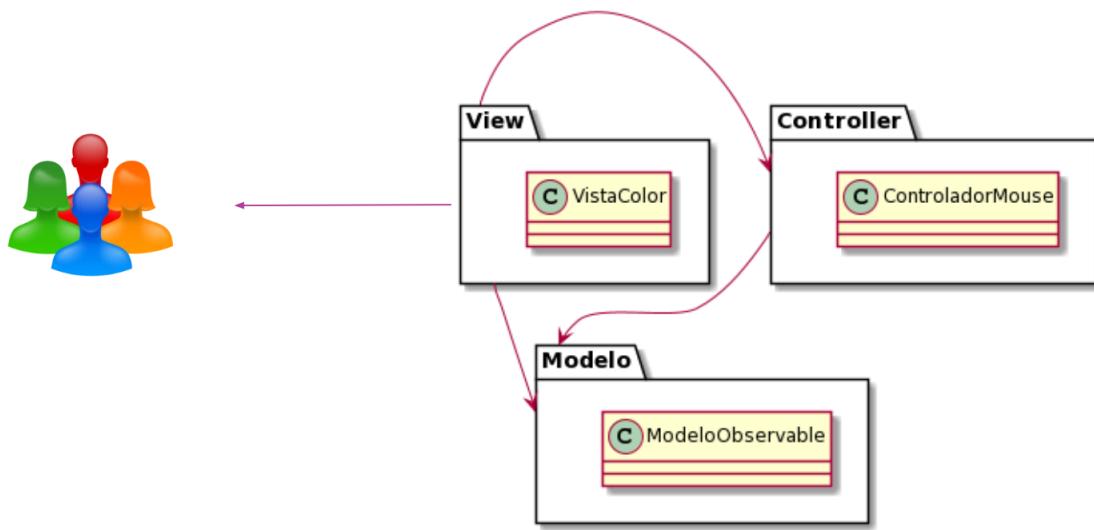
¿Cómo planea hacerlo?

- Procastinar decisiones no esenciales → (como vienen los datos → json? xml?...).
- Cohesión → Necesitamos bajo acoplamiento entre las partes de The Clean Architecture.
- Escritura de pruebas
- Diseño (UI UX paradigma)

Modelo - Vista - Controlador (MVC)

Tres tipos de objetos:

- Modelo: lógica y entidades de negocio o dominio de nuestra aplicación.
- Vista: formas en que los objetos del modelo se muestran y representan al usuario.
- Controlador: definen cómo la interfaz de usuario reacciona a las acciones del usuario.



*Obs: La flecha significa contiene (como atributo o parametro).*

## Patrón Observer

### Intención:

En criollo: Permite a un objeto ser “observado” (sujeto) por un conjunto de otros objetos (observador).

- Definir una dependencia uno-a-muchos entre objetos de manera que cuando un objeto cambie su estado, todas sus dependencias sean notificadas y actualizadas automáticamente.

### Motivación:

- Mantener la consistencia entre objetos que dependen entre sí, reduciendo el acoplamiento y preservando la reusabilidad.

En criollo: Cuando el sujeto sufre un cambio en su estado, “notifica” a los observadores a través de un método especial para ello.

Ejemplo: [https://github.com/fiuba/algo3\\_ejemplo\\_mvc\\_colores](https://github.com/fiuba/algo3_ejemplo_mvc_colores)

## Clase Complementaria: Persistencia

Un ambiente OO debe permitir que los objetos persistan, para mantener su vida mas allá de la vida de la aplicación.

Este concepto nos permite que un objeto pueda ser usado en diferentes momentos a lo largo del tiempo, por el mismo programa o por otros, así como en diferentes instalaciones de hardware en el mismo momento.

Un **objeto persistente** es aquel que conserva su estado en un medio de almacenamiento permanente, pudiendo ser reconstruido por el mismo proceso que lo generó u otro, de modo tal que al reconstruirlo se encuentre en el mismo estado en que se lo guardó.

Al objeto no persistente lo llamaremos **efímero o transitorio**.

Tipos de persistencia:

- Nativa → Provista por la plataforma (ej: Java, Smalltalk).
- No nativa → A través de una biblioteca externa o programada a mano.

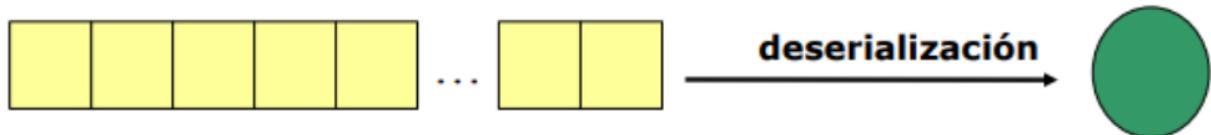
Lo más habitual es que los objetos orientados no soporten auténtica persistencia, si no, una variante que exige guardar y recuperar objetos de forma activa.

## Serialización

Es el proceso que consiste en convertir la representación de un objeto en un **stream** (flujo o secuencia) de bytes.



Reconstruir un objeto a partir de un stream de bytes se denomina **deserialización**.



## Formatos de serialización

- Formato propietario:
  - Mas eficiente en términos de uso de almacenamiento y tiempo de traducción.
  - Solo sirve para comunicar aplicaciones basadas en la misma plataforma.
- Uso de lenguaje estándar:
  - ej: JSON
    - Es le más popular de intercambio de información.

*Obs: Java posee serialización automática en formato propietario. Para trabajar con JSON hay bibliotecas.*

## Persistencia y serialización

- Para persistir primero debo serializar.
- Serializar no implica necesariamente persistir.
- Otras acciones luego de serializar:
  - Enviar por red

- Mantener en memoria
- Enviar a una impresora, etc.

## Persistencia No Nativa

Queda en manos del programador:

- Cuestiones de diseño
  - Responsabilidad
    - cada clase sabe como persistirse
    - Existe un gestor externo que sabe como persistir las clases
  - Formato
    - Binario, texto plano, texto jerárquico(XML)
  - Identidad de los objetos
    - Referencias circulares, duplicación de objetos, etc.
- Mayor versatilidad, pero...
- Mayor complejidad

## Clase Complementaria: Conurrencia

¿Por qué concurrencia?

- Performance
- Tiempo de respuesta al usuario
- Tiempo de ejecución de una aplicación
- “Mundo Paralelo”
- Aprovechar el hardware

## Conurrencia vs Paralelismo

*“Conurrencia es tratar de lidiar con muchas cosas a la vez. Paralelismo es hacer muchas cosas a la vez.” Rob Pike*

**Concurrencia** es la composición de la ejecución de “cosas” independientes. Es sobre la estructura de los problemas, descomponer tareas en tareas mas pequeñas.

**Paralelismo** es ejecutar tareas en simultáneo.

⇒ Concurrencia no implica Paralelismo

## Thread

- Secuencia independiente de instrucciones ejecutándose dentro de un programa.
- Función o clase que se ejecuta de manera concurrente.

## Problemas

- **Race Condition:** Se da cuando varios threads pueden acceder a recursos compartidos (código). El resultado del programa depende de cómo se intercalen los threads.
- **Critical Section:** Sección de código que necesita ser ejecutada en forma atómica por un solo hilo a la vez.

*Obs: cuando descuidamos la Critical Section se produce la Race Condition.*

## Solución (Sincronización)

- Locks: Se basa en el uso de una variable de exclusión mutua (mutex).
- Monitores: Objetos thread-safe, sus métodos están sincronizados (mutex).
- Conditional variable: Mecanismo de bloqueo con una señalización.
- Semáforos

## Mas Problemas - DeadLock

Aparece cuando entre dos o más threads uno obtiene un recurso y no lo libera generando un bloqueo.

## Consideraciones

- Debuggear es complicado.
- Agregar `println()` puede alterar el resultado. (porque influye en el tiempo de ejecución)
- Demasiada sincronización perdemos la ventaja de usar threads.
- Poca sincronización genera errores difíciles de detectar.

## Clase Complementaria: Interfaces de Usuario

Algunos problemas del mal diseño:

- Accidentes: Los sistemas mal diseñados ponen en peligro al usuario. Desde incidentes en aeronáutica, hasta accidentes hogareños y lesiones.
- Costo de soporte: A mayor dificultad de aprendizaje y operación, más se necesita invertir en capacitar a los usuarios, y en atender sus reclamos.
- Menos usuarios: De existir alternativas, los usuarios optarán por ellas.

## Human-computer Interaction / HCI / CHI

Área de la informática y ciencias de la computación que estudia cómo los humanos interactuamos con la tecnología. Nucleados bajo el grupo de interés SIGCHI dentro de la ACM, involucra varias disciplinas. Surgió a fines de los 70s



**Arte**

**Ciencias de la Computación**

**Diseño**

**Economía**

**Ergonomía**

**Ingeniería**

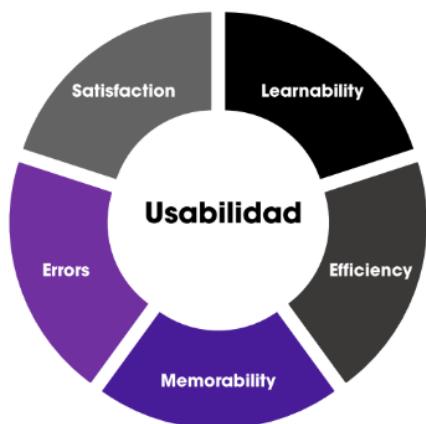
**Psicología**

**Sociología**

## Usabilidad

- Es el GRADO de efectividad de la interacción entre operadores y sus máquinas (Nielsen J.)
- Atributo de calidad que mide cuán fácil es usar una interfaz.

### Atributos de usabilidad



**Learnability/Aprendizaje:** ¿Qué tan fácil es para los usuarios hacer tareas básicas desde el primer uso?

**Efficiency/Eficiencia:** una vez que aprendieron el diseño, Qué tan rápido pueden completar tareas?

**Memorability/Memorización:** Cuando los usuarios vuelven a la interfaz después de un tiempo sin haberla usado, Qué tan fácil pueden volverse performantes nuevamente?

**Errors/Errores:** Cuántos errores cometen los usuarios, qué tan severos son, y cómo se pueden recuperar de ellos?

**Satisfaction/Satisfacción:** Qué tan placentero es usar la interfaz?

[Nielsen & Norman, 2012]

## **Gamificación o lidificación**

Uso de elementos del juego en contextos no lúdicos

Algunas técnicas de gamificación:

- Achievements (recompensas)
- Leaderboard (rankings)
- Comparación (ej: con otros jugadores)
- Puntos o bienes virtuales

¿Para qué entonces hacer productos usables?

- Reducir o eliminar costos de soporte y entrenamiento
- Aumentar la productividad o eficiencia del usuario
- Cumplir con estándares de calidad, de haberlos
- Reducir la carga física y cognitiva o mental
- Incrementar la satisfacción

## **User Experience UX**

Área que estudia todos los aspectos de la interacción de un usuario con el servicio, producto o software. No se limita a la UI (User Interface o interfase de usuario) o diseño visual solamente.

Abarca además procesos, estrategia del contenido y comunicación, usabilidad, investigación con usuarios y diseño de la arquitectura.

Una buena experiencia de usuario...

- Satisface los requerimientos de un usuario final.
- Genera productos que sean deseables de tener y utilizar.
- Involucra servicios de muchas disciplinas, incluyendo ingeniería, marketing, diseño industrial y visual, y diseño de interfaces.

## Algunos roles ligados a UX

- UX Researcher: investigación con usuarios, análisis de datos
- UX Designer: diseño de interacciones
- UX Writer: escritura del contenido y comunicación textual
- Visual designer: marca, diseño visual y gráfico
- UX Strategist: definición de estrategia general UX
- UI Engineer: frontend developer. Programación de la interfaz de usuario (UI)

## Accesibilidad

Capacidad de que un artefacto/sistema esté diseñado para la mayor cantidad de usuarios posible, con y sin discapacidad y otras condiciones.

Algunos métodos de diseño:

### 1. **User-centered design (HCD/UCD)**

Es un enfoque del proceso de diseño que toma en cuenta los requerimientos, limitaciones y necesidades de los usuarios en toda las etapas



#### **Involucra al usuario**

están involucrados en el proceso.  
Se empieza entendiendo la necesidad o los problemas de los usuarios.



#### **Iterativo**

Se mide y evalua la experiencia **con usuarios, iterando** hasta lograr un diseño mejorado a través de prototipos y evaluaciones.

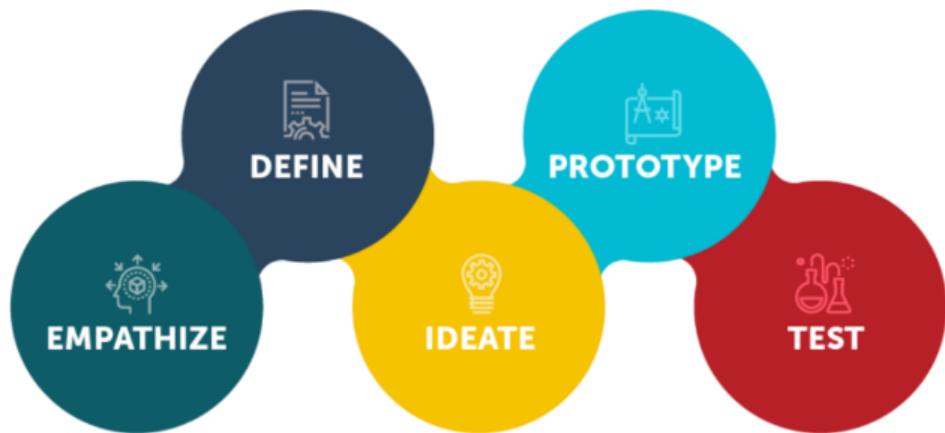
Proceso:



## 2. Design Thinking

Es un proceso iterativo con el objetivo de entender al usuario, desafiar asunciones, y redefinir problemas para identificar estrategias alternativas y soluciones.

Plantea 5 fases que pueden ser no secuenciales: empatizar, definir, idear, prototipar y testear.



## Etapas en el diseño centrado en el usuario

### 1. Investigación con usuarios (UX RESEARCH)

Conocimiento acerca del usuario y problema.

- Cual es la tarea?
- Objetivos
- Limitaciones
- Deseos

Algunos métodos de investigación con usuarios son: Entrevistas; Encuestas; Focus group; Diarios; Analíticas (I); Observaciones; Contextual inquiry; Etnografía; Exploración; Lectura de registros (I).

**Análisis de datos:** Una vez que se recolectan los datos, se asignan códigos, se buscan categorías y patrones, y se analiza el conjunto. En el análisis cualitativo el foco es entender por qué pasa. En el cuantitativo, entender qué pasa y en qué dimensión.

## 2. Diseño de prototipos

Distintos tipos de prototipos:

- De papel: rápido, económico, ideales para validar ideas iniciales.
- Wireframe: baja definición, sin colores ni fuentes, posee contenido y su organización.
- Mockups: tienen color, sin funcionalidad.
- De baja resolución: hecho con herramientas, tiene todas las interacciones, puede tener animaciones, sin funcionalidad real.

## 3. Evaluación o prueba de usabilidad

Son prácticas con el objetivo de aprender y medir con los usuarios la facilidad de uso de un sistema.

- En sesiones individuales, los usuarios ejecutan tareas en diversos escenarios mientras se observa y conversa para entender su visión.

- Permite identificar problemas de forma temprana, encontrar oportunidades de mejora, ó bien medir la usabilidad.
- Las variables usuales que se miden son éxito, tiempo, errores cometidos, satisfaccion, y dificultad percibida.
- Se eligen usuarios representativos.
- Con 5 usuarios se pueden detectar hasta el 85% de los problemas.

## Nielsen's Heuristics

Son principios generales de diseño de interacción que nos permiten encontrar mejoras en un sistema o sitio web.

1. Visibilidad del estado del sistema: El Sistema siempre tiene que mantener a los usuarios informados de lo que está pasando, y dar una respuesta en tiempo razonable.
2. Conexión sistema-mundo real: El Sistema debe hablar el idioma del usuario, con frases que conoce y son familiares y no jerga técnica. Siga las convenciones del mundo real, y haga que la información aparezca en el orden natural y lógico.
3. Control de usuario y libertad: Soportar el hacer y deshacer (undo). Los usuarios suelen elegir opciones por error y necesitan una salida delimitada para abandonar ese estado.
4. Consistencia y estándares: Comandos: misma acción, mismo efecto. Tareas similares se manejan igual.
5. Prevención de errores: Un diseño que previene errores desde el inicio es mejor que unos buenos mensajes de error.  
Elimine condiciones que conducen a errores, o verifíquelas y preséntele a los usuarios una confirmación antes de ejecutar la acción.
6. Reconocimiento sobre memoria: Minimice el uso de memoria del usuario haciendo que objetos, acciones y opciones estén visibles. El usuario no debería recordar información de una interfaz en otra.  
Las instrucciones de uso deberían ser accesibles en le momento necesario.
7. Flexibilidad y eficiencia de uso: Poner atajos a las operaciones o acciones más usadas. Un experto debería realizar las operaciones frecuentes rápidamente.

8. Estética y minimalismo: La interfaz no debería contener información no relevante o poco usada. Cada unidad extra de información compite con la relevante, y degrada su visibilidad.
9. Ayudar a usuarios a reconocer, diagnosticar y recuperarse de errores: Los mensajes de error deberían mostrarse en lenguaje plano, sin código, indicar el problema y sugerir una solución.
10. Ayuda y documentación: Aunque es mejor que el sistema sea usado sin documentación, puede ser necesario proveer ayuda. Debería ser fácil de buscar y enfocada en la tarea del usuario, listar pasos concretos y no ser muy extensa.

## Recomendaciones

- Tipografía: Elegir tipografías “cuidadas” en el texto para mejorar legibilidad.
- Color
  - Definir una paleta de hasta 5 colores.
  - No usar el color como única forma de informar: Personas con daltonismo (ceguera al color) podrían no percibirlo. La interpretación es puramente cultural.
  - Elegir colores con buen contraste: Personas con baja visión o en entornos muy luminosos podrían no ver la información.
- Organización
  - Diseñar el texto para el “picoteo” / scam : Asegurarse de que lo más importante esté al inicio.
  - Ley de Miller: El número de oro 5 +/-2 para los elementos en un grupo.
  - Formularios: Agrupar los elementos similares, mostrar errores y campos requeridos.
  - Ley de Hicks: Reducir la cantidad de opciones. El tiempo para tomar una decisión depende de cuántas opciones se muestren.
  - Corolario de Ley de Fitts: Agrandar los elementos principales.
  - Efecto Von Restroff: Aislar los elementos a destacar. Cuando hay muchos elementos similares, el que más difiere es el más recordado

- Serial Position Effect: Disponer elementos importantes al principio y al final

## Resumen de autor: Patrones de Diseño

Un patron de diseño es la manera de resolver un problema. Debe cumplir al menos con los siguientes objetivos: estandarizar el lenguaje entre programadores, evitar perder tiempo en soluciones a problemas ya resueltos, crear codigo reusable.

Los patrones de diseño pueden clasificarse en:

- Creacionales → procuran independizar al sistema de como sus objetos son creados y/o representados.
- Organización
- Control de acceso
- Variaciones de servicios
- Extensiones de servicios
- Descomposición estructural

## Patrones Creacionales

### 1. SINGLETON

Propósito:

- Garantiza que una clase solo tenga una instancia y proporciona un punto de acceso global a ella.

Solución:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.
- Declara el constructor de clase como privado para que no sea instanciable directamente.

Consecuencias:

- Acceso controlado a la instancia
- Se logra el objetivo pero a cambio de ensuciar la clase.
- Es facil pero intrusivo.

## 2. MULTITON

Garantiza que tenga varias instancias conocidas. Se implementa con un mapa (identificados, instancia).

## 3. ABSTRACT FACTORY

Proposito:

- Proporciona una interfaz para crear familias de objetos relacionados o que dependan entre si, sin especificar sus clases concretas.

Solución:

- Crear una clase factory que sea abstracta que provea una interfaz comun en familias.
- Crear clases factories que hereden de factory, y que implementen métodos creando las instancias correctas.

Consecuencias:

- Independencia de clases concretas.
- Permite intercambio de familias de objetos de manera rápida y transparente.

# Patrones de Organización

## 4. COMMAND

Proposito:

- Encapsula una petición en un objeto, permitiendo parametrizar a los clientes con diferentes peticiones, hacer cola o llevar registro y poder deshacer las operaciones. Desacopla el código que solicita un servicio del que lo presta.

Solución:

- Crear una clase abstracta/interfaz con un solo método execute()
- Clase descendiente implementara el método.
- Para invocar el método se instanciara una de las clases y se invoca el método execute().

Consecuencias:

- Permite mantener referencia a métodos.
- Desacopla el objeto que invoca la operación de aquel que sabe como realizarla.
- Facil de añadir ordenes nuevas.
- Las ordenes son objetos de primera clase pueden ser manipulados y extendidos.

## Patrones de Control de Acceso

### 5. PROXY

Proposito:

- Proporciona un representante de otro objeto para controlar el acceso a este.

Solución:

- Jerarquía entre el objeto original y el objeto proxy.
- En el objeto proxy hay una referencia al original.
- Se definen las llamadas en el proxy.
- Retrasa el coste de creación e inicialización hasta que sea necesario.

Consecuencias:

- Agrega funcionalidad de manera transparente a la aplicación.
- Realiza optimizaciones, oculta complejidad.

### 6. FACADE

Proposito:

- Interfaz unificada para un conjunto de interfaces de un subsistema. Es de alto nivel y el subsistema es más fácil de usar.

Beneficios:

- Oculta al cliente el subsistema, haciéndolo más fácil de usar.
- Debil acoplamiento entre subsistema y clientes.
- No impide que se usen clases del subsistema. Así se puede elegir entre facilidad y generalidad.

## Patrones de Variaciones de Servicios

### 7. STRATEGY

Un mismo objeto debe poder tener un comportamiento que es determinado en tiempo de ejecución.

Solución:

- Delegar comportamiento en otro objeto.
- Jerarquía con distintos comportamientos.
- Inyectar el comportamiento al objeto a través de un método o constructor.

Consecuencias:

- Se eliminan los condicionales.
- Comportamientos agrupados en familias.
- No es fácil aislar el comportamiento.

### 8. TEMPLATE

Es el esqueleto de un algoritmo, delegando en las subclases algunos pasos permite que las subclases ... rediseñan otros ... pasos sin cambiar su estructura.

Aplicabilidad:

- Implementar partes que no cambian y dejar que las subclases implementen el comportamiento variable.
- Para factorizar comportamiento repetido.

- Para controlar las extensiones de las subclases
- Técnica de reutilización de código.
- Extraen el comportamiento común de las clases de la biblioteca.

## 9. STATE

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

Aplicabilidad:

- El comportamiento de un objeto depende de su estado.
- Las operaciones tienen largas sentencias condicionales con muchas ramas que dependen del estado del objeto.

Beneficios:

- Localiza el comportamiento dependiendo del estado y lo divide en distintos estados.
- Explicita transiciones entre estados.
- El objeto estado puede compartirse.

# Patrones de Extensión de Servicios

## 10. DECORATOR

Asigna responsabilidades adicionales a un objeto dinámicamente proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

# Patrones de Descomposición Estructural

## 11. COMPOSITE

Compone objetos en estructura de árbol para representar jerarquías de parte-todo.