Unidad 1: Análisis y Diseño

Metodología de la Programación

Curso 2020-2021

© *Candi Luengo Díez , Alberto M. Fernandez Álvarez, Francisco Ortín Soler y José Manuel Redondo López*

Bibliografía

- **Programación orientada a objetos con Java. 6th Edición**
  David Barnes, Michael Kölling.
  Pearson Education. 2017

  Capítulo 15: Diseño de aplicaciones
  Capítulo 8: Diseño de clases

- **El Lenguaje Unificado de Modelado. Guía del usuario**.
  Grady Booch, Ivar Jacobson, James Rumbaugh.
  Addison-Wesley/Diaz de Santos, 1999.

- **Construcción de software orientado a objetos (2ª edición)**
  Bertrand Meyer. Prentice Hall, 2000.

Principales conceptos

- Análisis y Diseño
- Metodologías de desarrollo
- Notación **UML**
- Principios básicos para el diseño de clases
- Refactorización

DISEÑO DE APLICACIONES

Análisis vs Diseño

- **Análisis**
  - Decidir **qué** debe hacer la aplicación
    - Modelar el "mundo real", el "dominio"
      - Entender qué se necesita hacer y qué misión tendrá la nueva aplicación.
      - Entender cómo funciona lo que existe .
    - *¿Que tipo de datos del cliente necesito almacenar?*
- **Diseño**
  - Decidir **cómo** será la aplicación
    - Resolver los problemas técnicos
    - *¿Es apropiado usar un fichero XML para almacenar los datos del cliente?*

Análisis y diseño

- **Análisis**: **dominio del problema (Qué)**
  - Todas las entidades deben ser entendidas por el usuario (Empleado, Departamento, Nómina…)
  - No deben aparecer los problemas técnicos (se dejan para la fase de diseño)
- **Diseño**: **dominio de la solución (Cómo)**
  - Figuran entidades técnicas, estructuras internas que utilizaremos para resolver el problema pero que el usuario no ve. (Controller, Façade, Iterator, Translator…)
- Los programas se crean siguiendo una **metodología** (UML)
- Se necesita **una notación estándar** para describir el análisis y el diseño de un programa.
  - Los **arquitectos** usan **planos** para describir las construcciones.
  - Los **Ingenieros Informáticos** usan **diagramas** para describir el software.

Metodologías de desarrollo de software

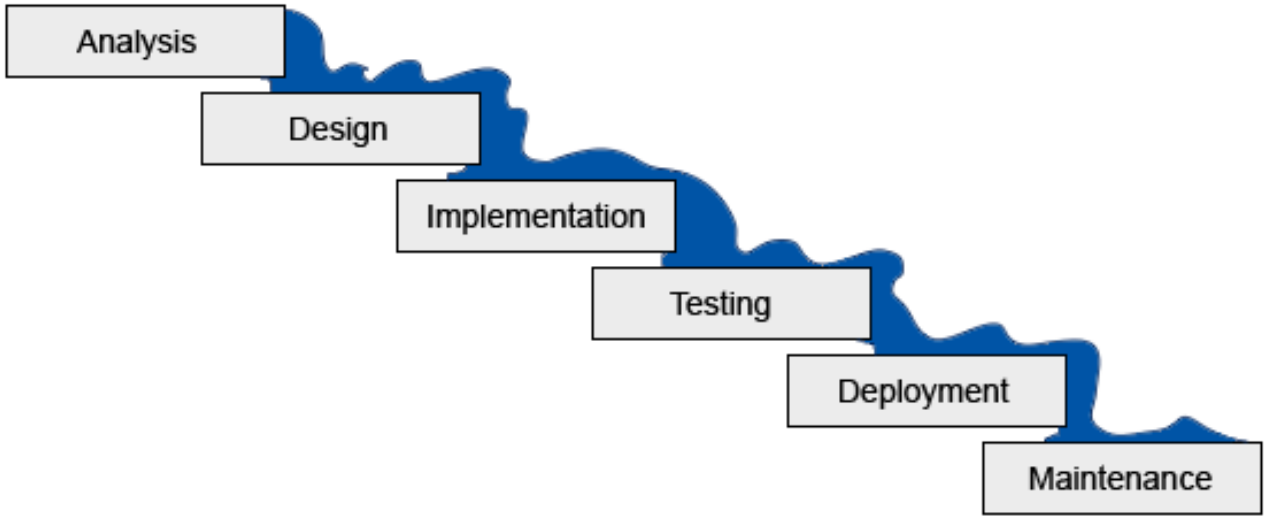
" Entorno usado para estructurar, planificar y controlar el proceso de desarrollo".

- Existen varios modelos:
  - **Cascada**
    - El desarrollo se lleva a cabo según una secuencia fija:

      Análisis del problema – diseño del software – implementación de los componentes del software – pruebas de unidad – pruebas de integración – entrega del sistema al cliente.
  - **Iterativo**
    - El desarrollo itera varias veces a través del ciclo:

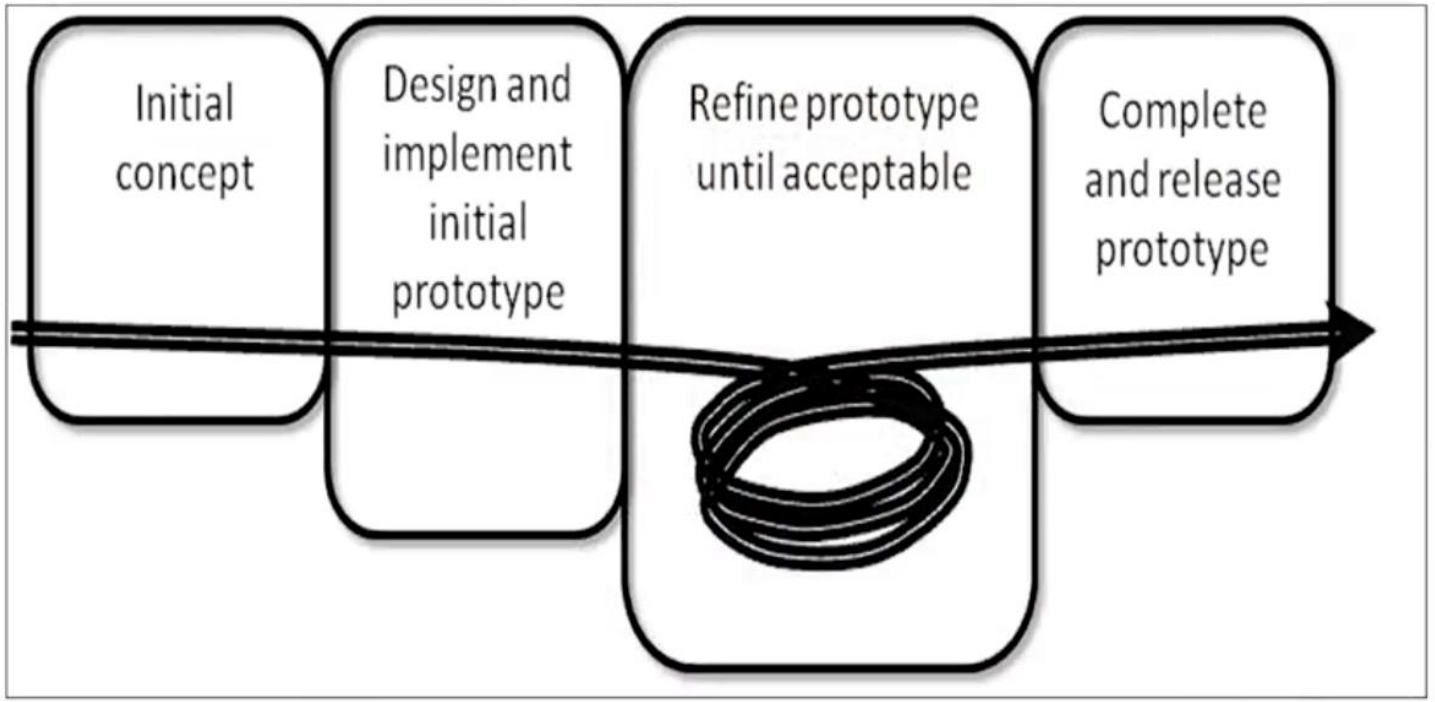
      Análisis – diseño – implementación prototipo – realimentación cliente.
  - **Incremental** *(Iterativo + Lineal)*
  - **Espiral** *(Incremental + análisis de riesgos)*
  - **Agil** *(Incremental)*

Modelo en cascada



- Muy simple de usar.
- Cada fase debe completarse antes de la siguiente.
- Adecuado para **proyectos pequeños** con requisitos bien conocidos.
- Desventajas:
  - Se ven los resultados tarde.
  - Los fallos se descubren en la fase de pruebas, demasiado tarde.
  - Muy arriesgado con proyectos medianos o grandes.

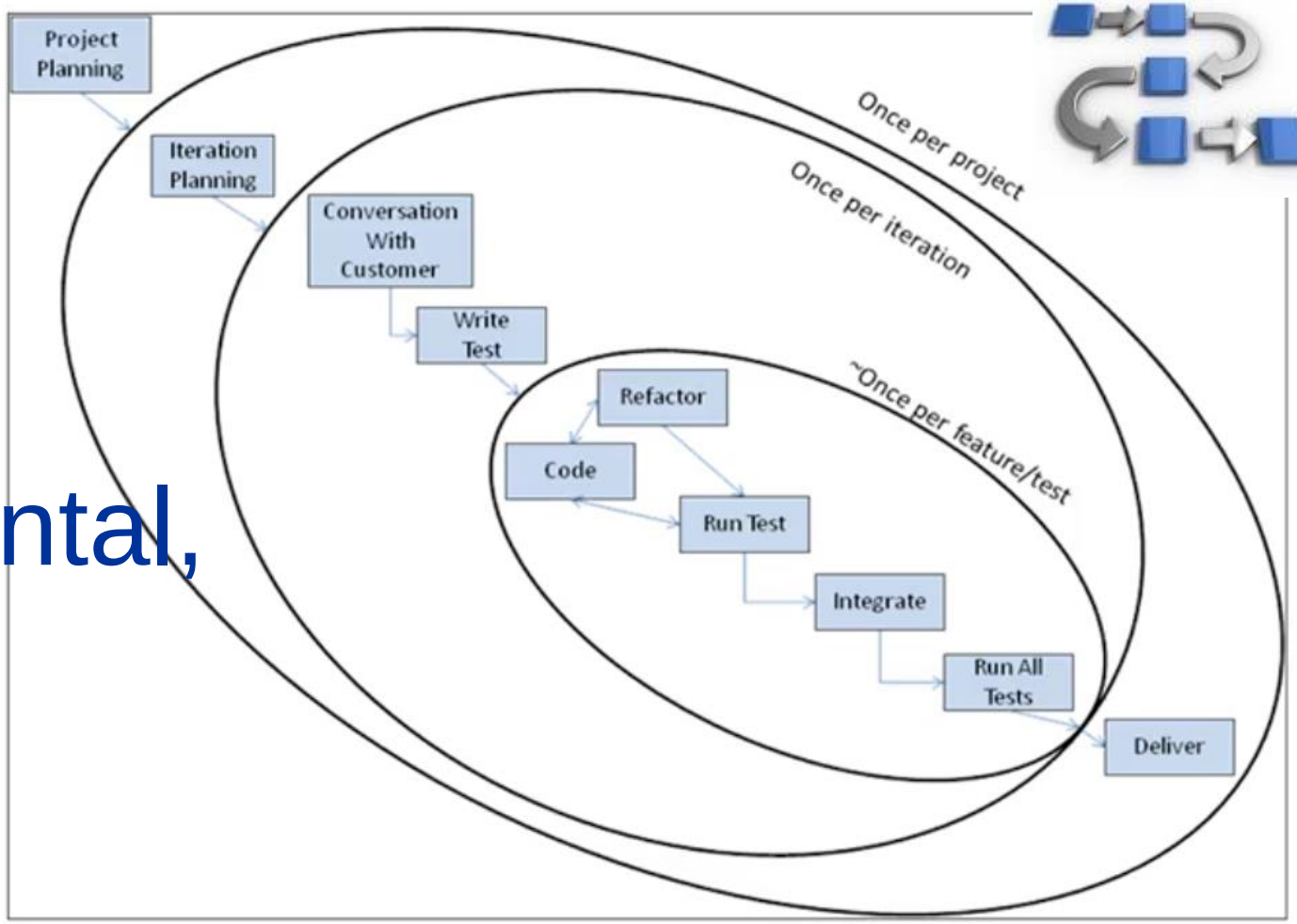
Modelo Iterativo (prototipado)



- Los usuarios participan activamente en el desarrollo.
- Los errores se pueden detectar mucho antes.
- Bueno para sistemas con interacción compleja del usuario.
- Desventajas:
  - Implementando y cambiando continuamente.
  - Puede comenzar con un análisis incompleto o inadecuado.

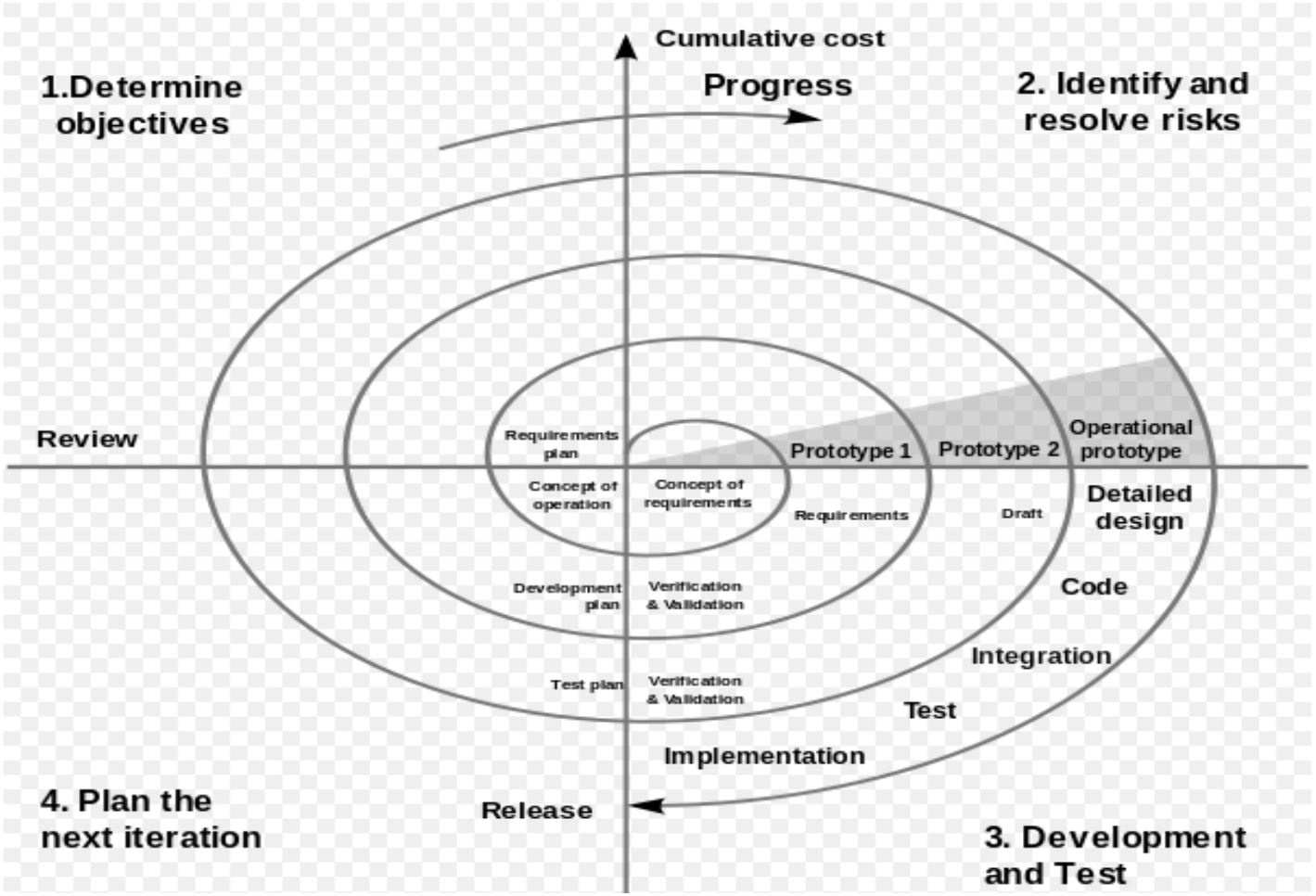
*** RUP (**Rational Unified Process) metodología iterativa usada en sistemas OO

Incremental, Agile



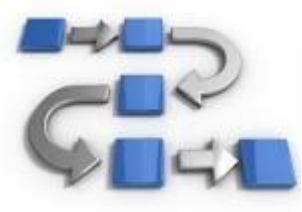
- Trabaja incrementalmente añadiendo pieza por pieza, pero cada pieza está completamente terminada.
- Flexible, permite cambiar los requisitos en la siguiente iteración.
- Más fácil de probar después de cada incremento.
- Desventajas:
  - Necesita una buena planificación y diseño.

Spiral



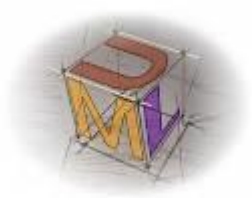
- Modelo incremental con más énfasis en el análisis de riesgo.
- Bueno para proyectos grandes y de misión crítica.
- El análisis de riesgos requiere una gran experiencia específica.
- El éxito del proyecto depende en gran medida de la fase de análisis de riesgos.

Metodologías de desarrollo de software



- Hay que dar **más importancia** al modelo de crecimiento del software que al modelo en cascada.
- El objetivo de las tareas irán encaminadas a:
  - El mantenimiento del software
  - Lectura del código (no solo de escritura)
  - Diseño con vistas a su ampliación
  - Codificación con vista a la legibilidad
  - La documentación
  - …
- Habrá programadores que vengan después de nosotros y tendrán que adaptar y ampliar el código.
- Hay que contemplar el software como una entidad que crece, cambia y se adapta continuamente, para escribir un buen código.

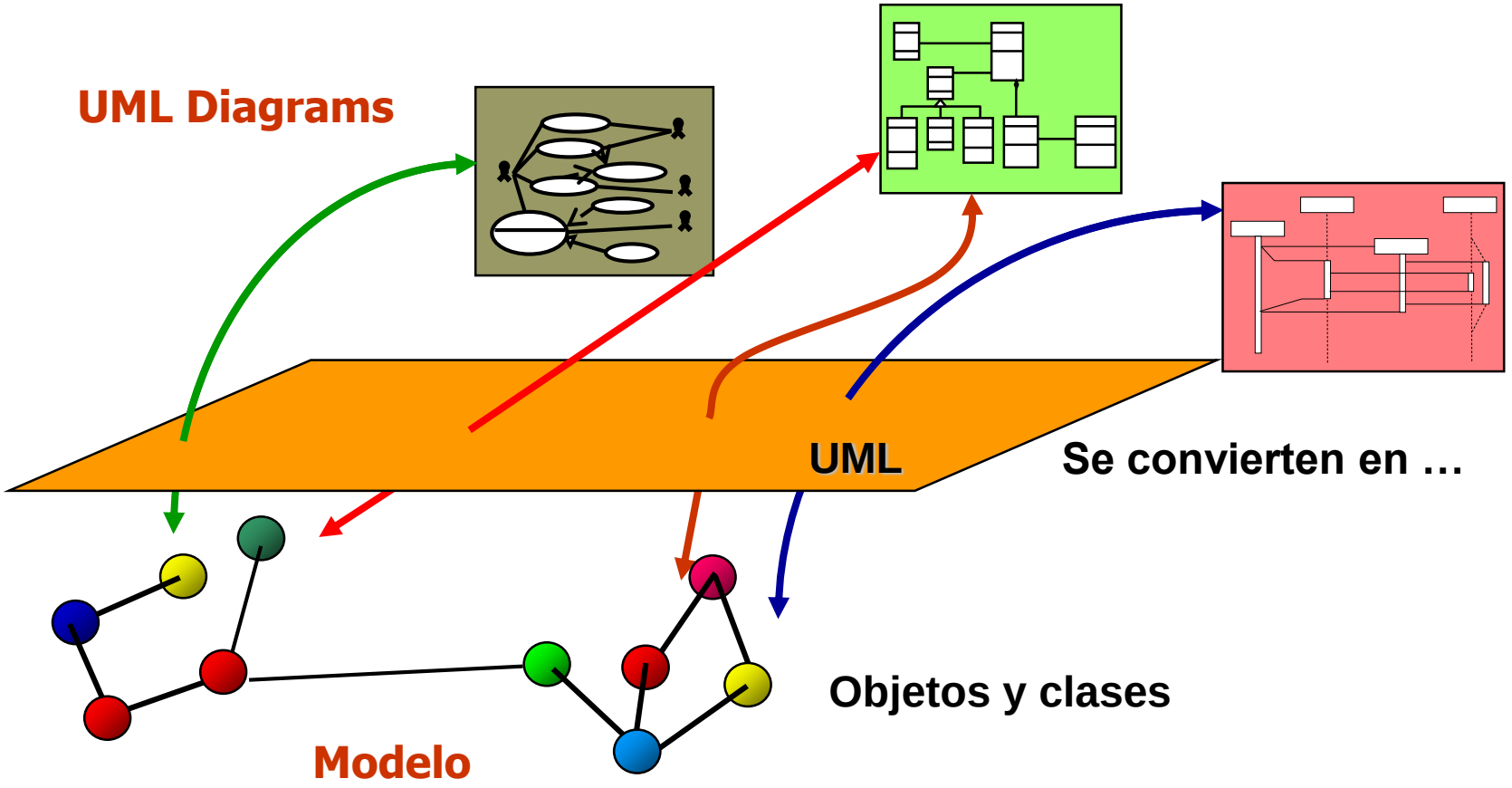
Notaciones de análisis y diseño



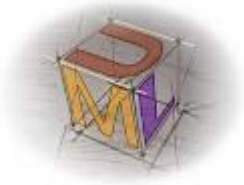
- **Lenguaje natural**
  - Ambiguo y poco práctico
- **Notaciones visuales** (como **UML**)
  - Medio de comunicación preciso y fácil de entender
    - Entre usuarios e ingenieros y entre ingenieros
  - De uso general en ingeniería (planos)
  - Existencia de herramientas (p.ej. Enterprise Architect)
- Una notación **NO** es un proceso de desarrollo o metodología.
  - Aprenderemos a construir software utilizando técnicas de ingeniería para cumplir los requisitos de usuario.
  - **ERES UN INGENIERO** (piensa una solución, crea el modelo, impleméntalo, pruébalo, despliégalo).

Vistas en UML

- Con UML podemos representar diferentes vistas de un modelo
  - Hay diferentes proyecciones del sistema
  - Las vistas deben ser coherentes, sin inconsistencias



Diagramas UML



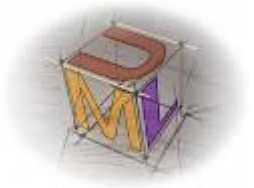
- Estructurales (estáticos)
  - **Clases**
  - **Objetos**
  - Componentes
  - Despliegue
  - Paquetes
  - De estructura compuesta
- Comportamiento
  - Casos de uso o escenarios
  - **Secuencia**
  - **Máquina de estados**
  - **Actividad**
  - Comunicación
    - Colaboración
  - …

enfatizan en los **elementos** **que deben existir** en el sistema, su **estructura** y **conexión**

enfatizan en **lo que debe suceder** en el sistema. Su **comportamiento dinámico**

Tranquilidad… En principio, usaremos sólo los que están marcados en azul

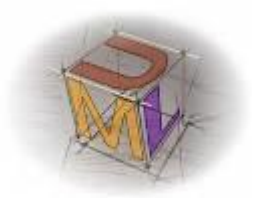
Notaciones de análisis y diseño



- Dudas sobre el **nivel de detalle**
  - Representar lo que resulte relevante
  - El detalle depende de lo que queramos trasmitir

  (detalle(análisis) < detalle(diseño) < detalle(implementación)
- Dudas sobre los **diagramas**
  - Elegir el más apropiado para trasmitir tu idea
  - Diagramas diferentes para las diferentes partes del modelo software (como los planos de electricidad de un edificio)
- Dudas sobre **análisis y diseño**
  - Los mismos diagramas puede utilizarse para ambos
    - Diferentes entidades a modelar (`problema` o `solución`)
    - Diferente nivel de detalle (detalle(análisis) < detalle(diseño))

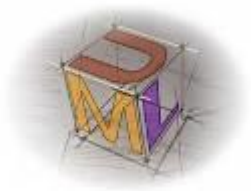
Modelo estático: **Diagramas de clases**



- Nombre de la clase
- Propiedades (Atributos)
- Métodos
- Tipos :
  - Propiedades
  - Parámetros
  - Valores retorno
- Visibilidad
  - Privada: -
  - Protegida: #
  - Pública: +
- Estereotipos: << >>

Diccionario
- fechaEdicion : Fecha - titulo : String - lenguajeOrigen : String - lenguajeDestino : String
+<<constructor>> Diccionario + traducir(palabra : String): String

Modelo estático: **Diagramas de clases**



Diccionario
- fechaEdicion : Fecha - titulo : String - lenguajeOrigen : String - lenguajeDestino : String
+<<constructor>> Diccionario + traducir(palabra : String): String

Se traduce a …

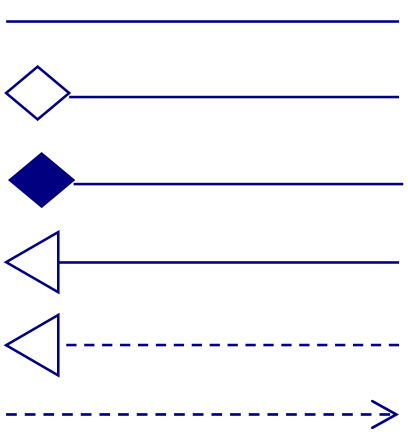
```
class Dictionary {
        private Date publicationDate;
        private String title;
        private String sourceLanguage;
        private String targetLanguage;

        public Dictionary() {
                //...
        }
        public String translate(String word){
                //...
        }

}
```

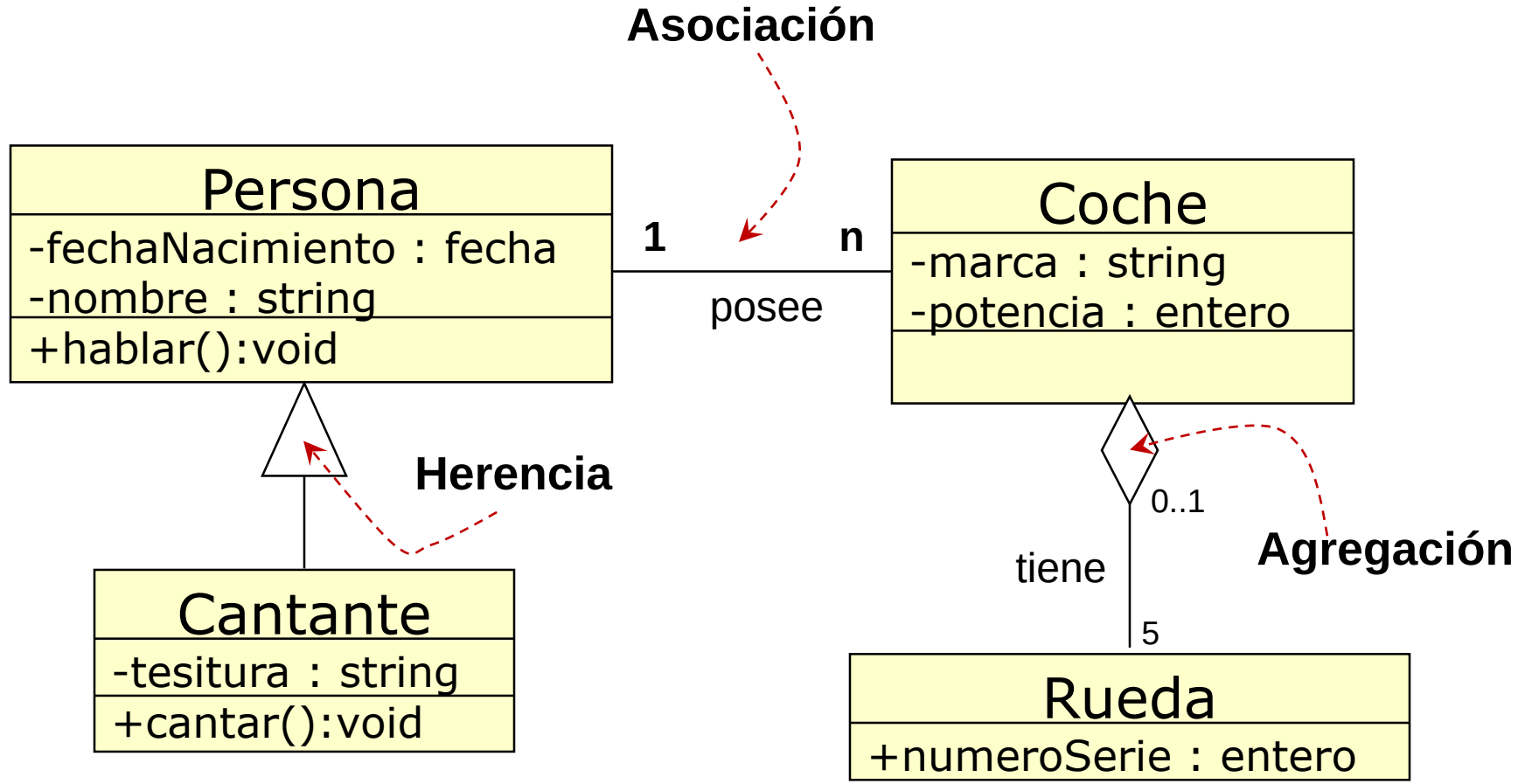
Modelo estático: **Relaciones**

- Relaciones entre clases
  - Asociación (relación)
    - Asociación simple
    - Agregación (*todo / parte*)
    - Composición (exclusivo *todo / parte*)
  - Generalización (herencia)
    - Realización (implementación)
  - Dependencia (uso)

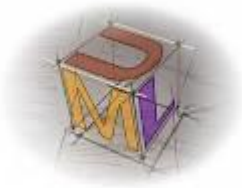


- Cada asociación tiene dos roles posibles
- Multiplicidad puede aparecer en cada extremo: `1`, `1..2`, `n`, `0..n`, `*` (0 ó más), `+` (uno ó más)
- Una asociación puede indicar navegabilidad, la dirección con una flecha

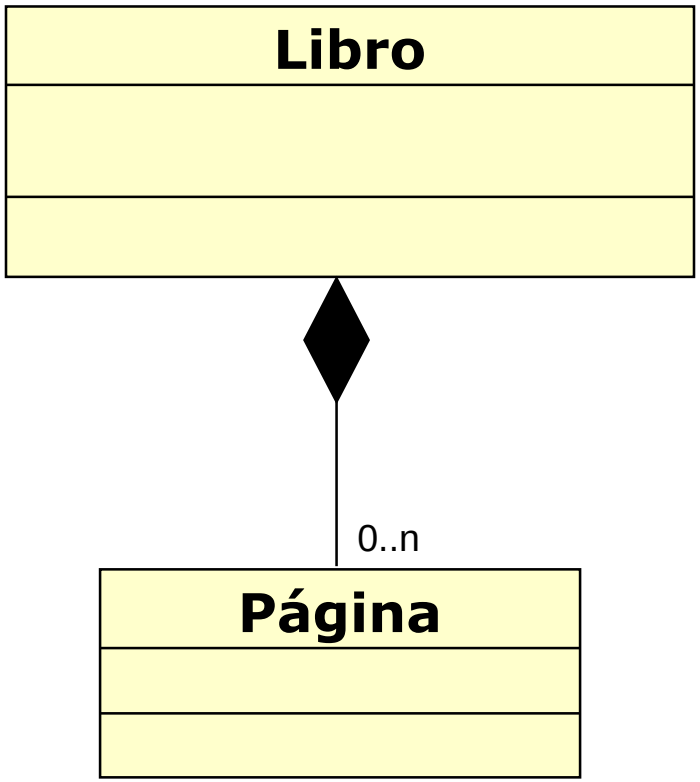
Modelo estático: **Relaciones**



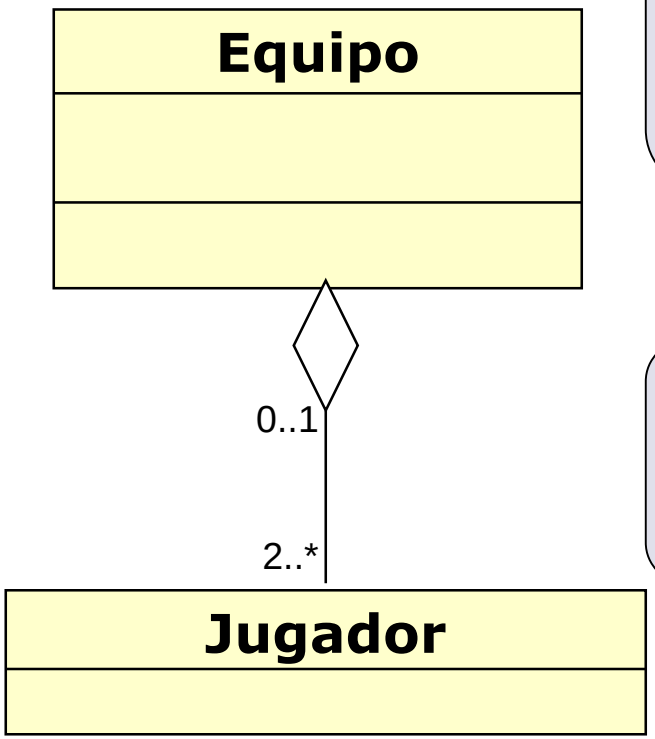
Modelo estático: Relaciones



Composición



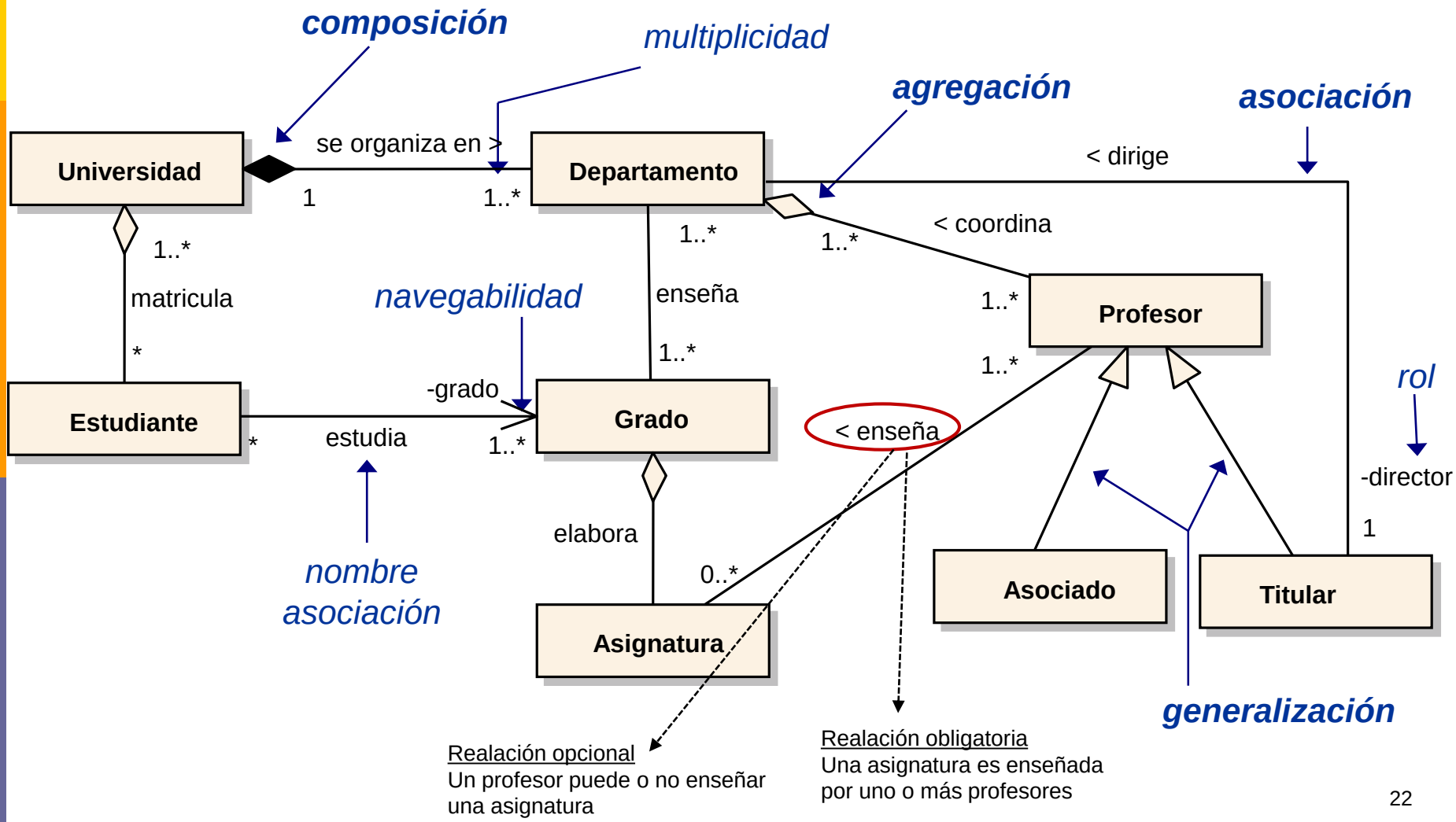
Agregación



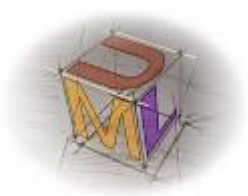
Agregación
los objetos "contenidos" pueden tener "vida" fuera del objeto que los contiene

En la composición **NO**

Modelo estático: relaciones



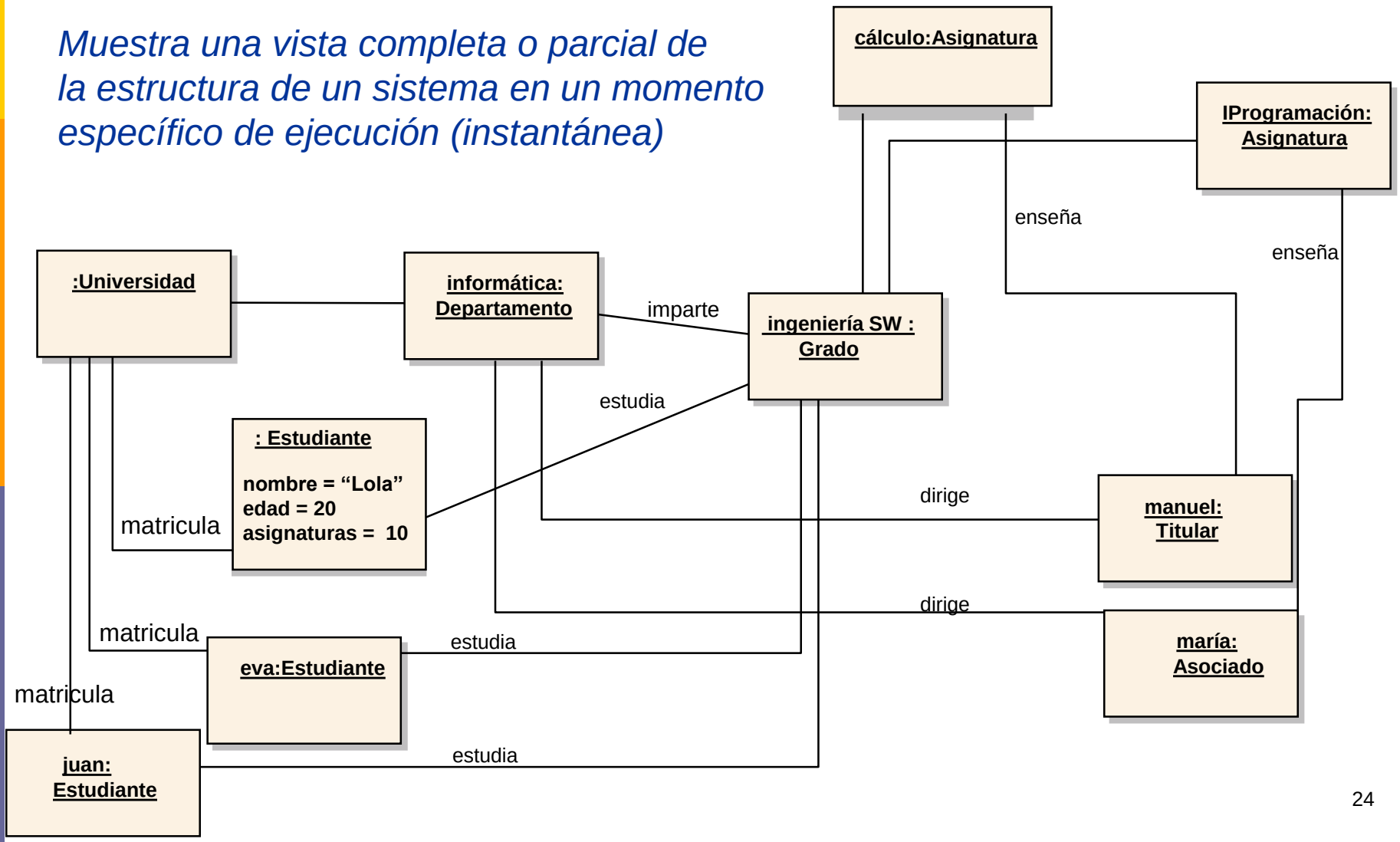
Modelo estático: **Relaciones**



- La agregación y composición son relaciones **todo/parte**
  - La composición es un tipo de agregación
  - La agregación es un tipo de asociación
- La composición tiene una dependencia más fuerte
  - Si el **todo** es destruido, sus partes también son destruidas.
  - No puede ocurrir la siguiente relación:

Modelo estático: **Diagramas de Objetos**

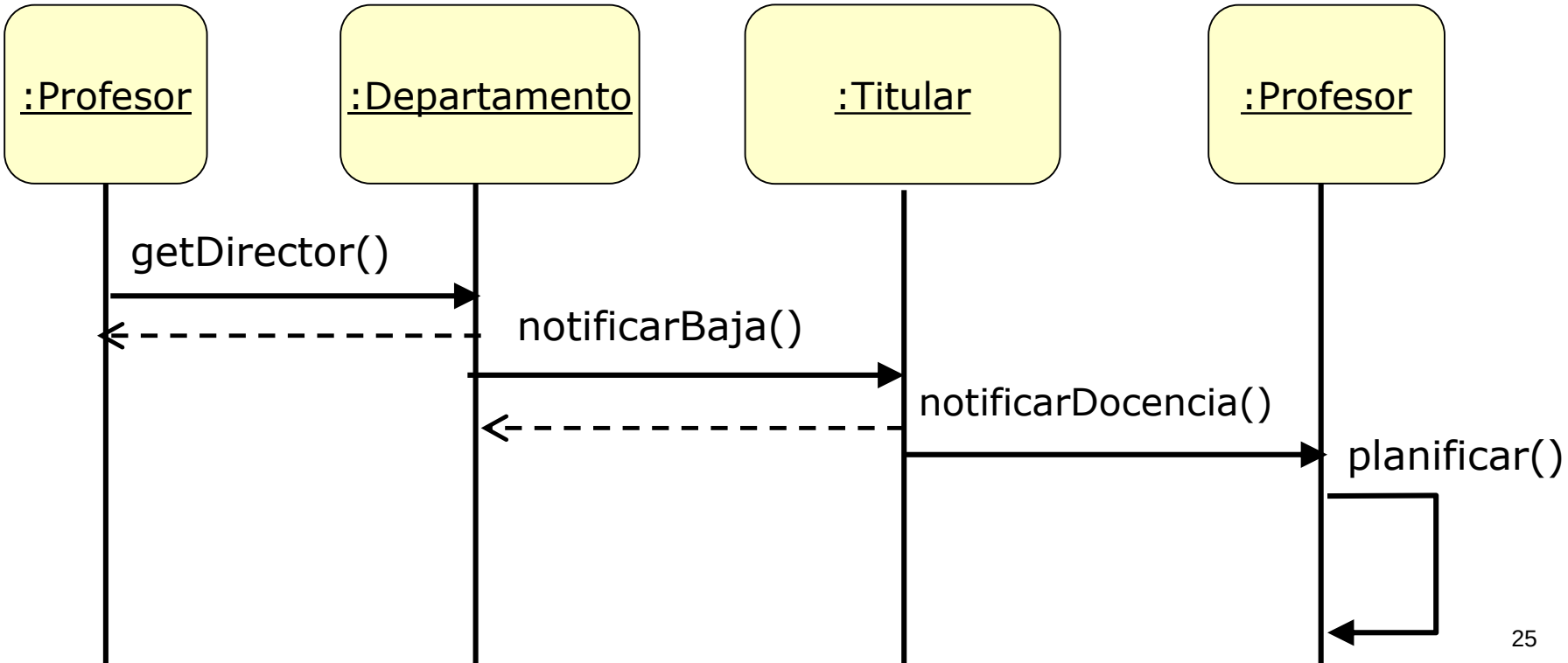
Muestra una vista completa o parcial de la estructura de un sistema en un momento específico de ejecución (instantánea)



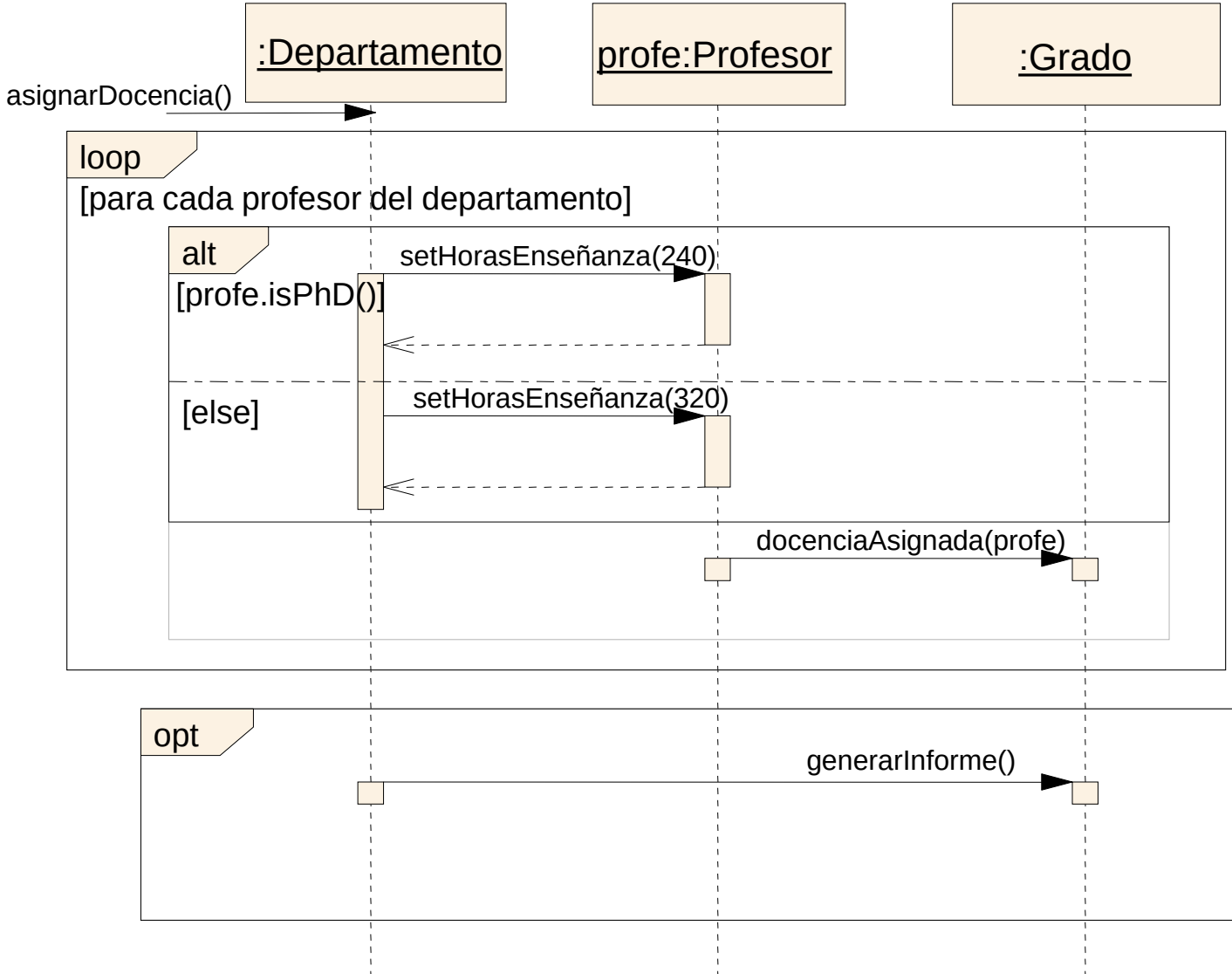
Modelo dinámico: **Diagramas de secuencia**

- Modela interacciones entre los **objetos** (no clases)
- Enfatiza el **orden temporal de los mensajes** (acciones)

Un profesor deja temporalmente la docencia por enfermedad y se pone en contacto con el Departamento que notificará la incidencia al profesor titular.
A continuación, el profesor titular buscará un profesor sustituto y este planificará el material docente.



Modelo dinámico: **Diagramas de secuencia**



El departamento asigna horas de docencia a cada profesor. Si es Profesor Doctor tiene 240 horas de docencia, en caso contrario, tiene 320 horas.

Una vez establecida la docencia, se asigna el profesor a un grado.

Mas tarde (opcional), el sistema genera un informe de la docencia.

Modelo dinámico: **Diagramas de estados o Máquinas de estados**

- Permiten modelar el comportamiento dinámico de parte del sistema.

- Principalmente, **el estado de los objetos de una clase** y los eventos que provocan cambios en esos estados.

- **Son de gran utilidad en ciertos campos de la informática:**
  - Sistemas en tiempo real: software para monitorizar procesos
  - Dispositivos: cajeros automáticos
  - Juegos: Doom, Half Life

Modelo dinámico: **Máquina de estados**

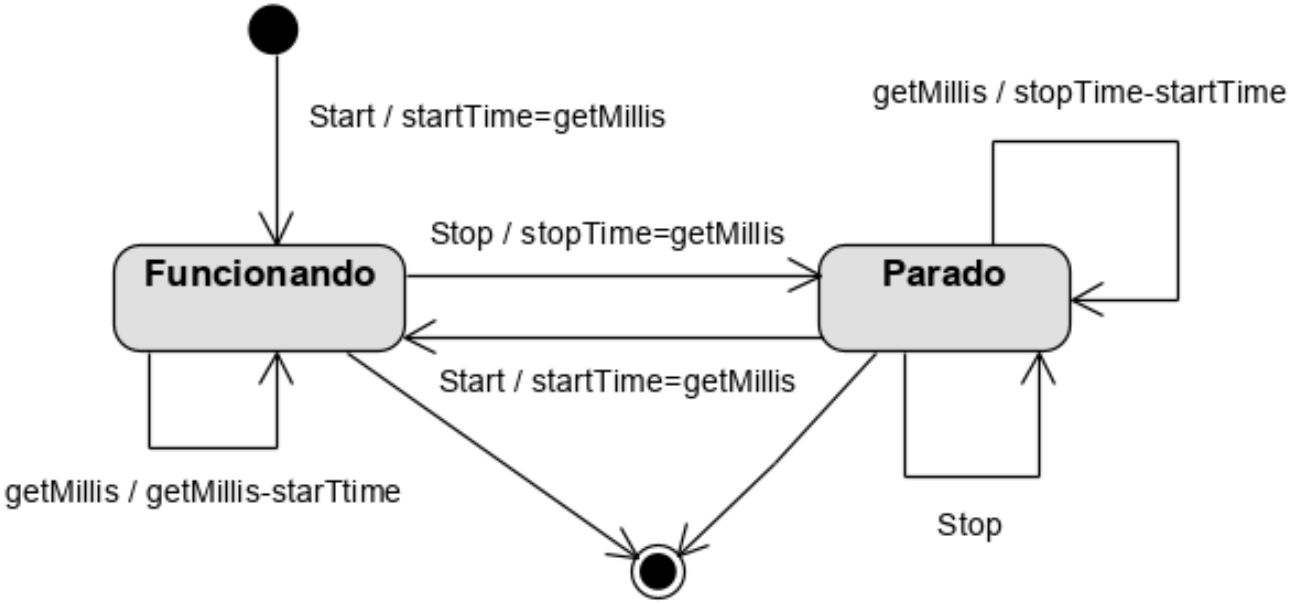
- **Consta de estados y transiciones**

Ejemplo

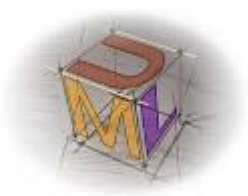
Cronometro
-startTime: int -stopTime: int
+start() +stop() +getMillis():int

- Estado inicial ●
- Estado Final ◉
- Estado E1
- Transición → **Evento[condición] / acción (es)**

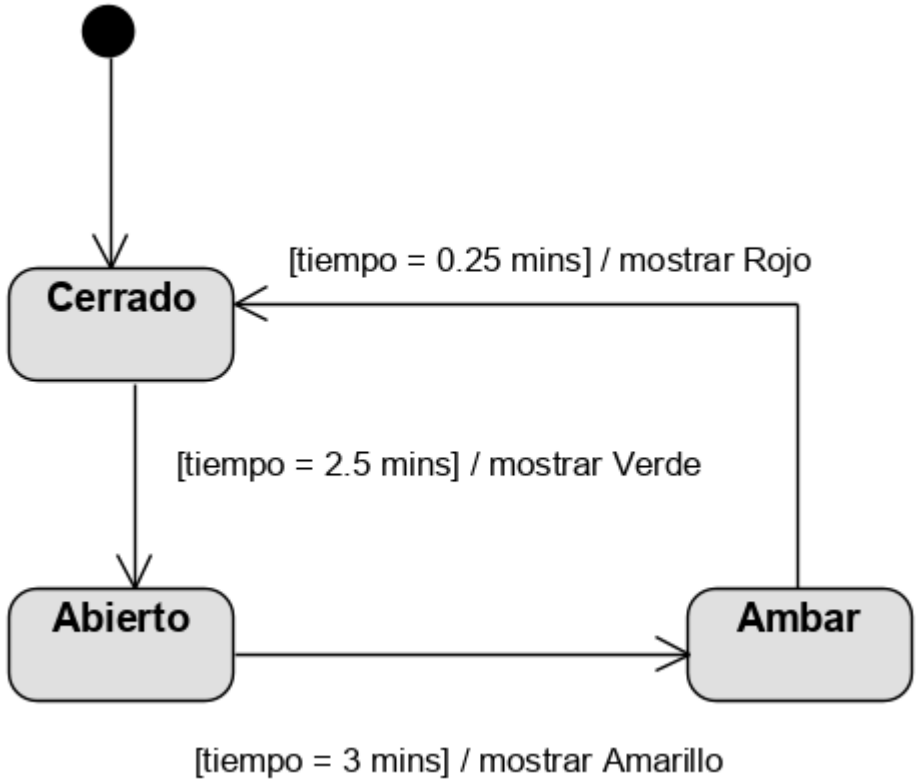
"Cuando se inicia un cronómetro, almacena su tiempo. Si se detiene, se calcula el tiempo transcurrido.
Si se reinicia, la hora de inicio se restablece. Si el cronómetro se detiene repetidamente, no se realiza ninguna acción"



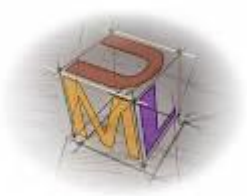
Modelo dinámico: **Máquina de estados**



Ejemplo: Semáforo

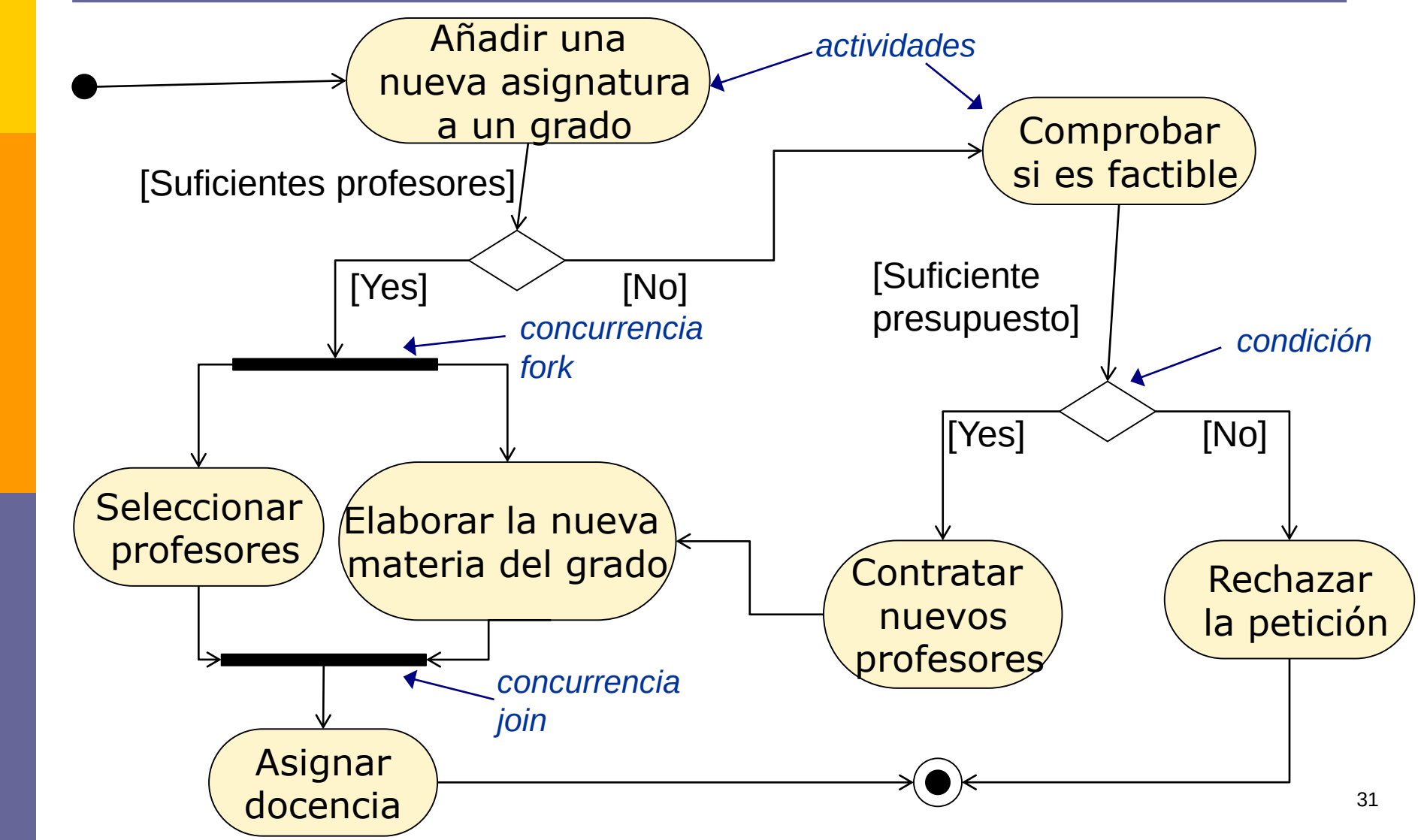


Modelo dinámico: **Diagramas de actividad**



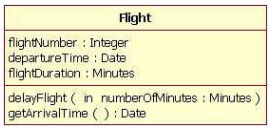
- Muestra los pasos de:
  - **Algoritmos**
  - **Procesos**
  - **Workflows**
  - **Escenarios multihilo**
- Cada paso esta reflejado como una actividad.
- **No es apropiado para modelar:**
  - Colaboración entre objetos (usar **diagramas de comunicación** y **diagramas de secuencia**)
  - Comportamiento dinámico de instancias de una clase (usar **diagramas de máquina de estado**)

Modelo dinámico: **diagramas de actividad**



DISEÑO DE CLASES

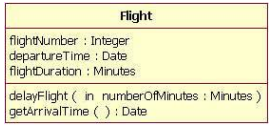
Diseñando clases



"Cómo escribir clases de manera que sean fácilmente comprensibles, fáciles de mantener y reutilizables"

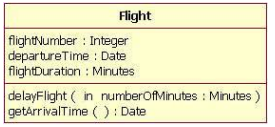
- Un buen diseño de clases permite un buen mantenimiento del código
  - **Fácil de modificar (*adaptive*)**. Se puede adaptar la aplicación a nuevos entornos.
  - **Fácil de extender (*perfective*)**. Se puede añadir nueva funcionalidad cuando se necesite.
  - **Fácil de depurar (corrective)**. Se pueden encontrar y resolver los errores.
  - **Más robusto (preventive)**. La aplicación puede hacer frente a errores durante la ejecución, es decir, continuar funcionando a pesar de los errores producidos.

Calidad del código



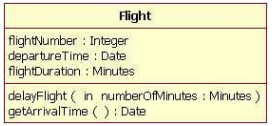
- Dos conceptos importantes para el diseño de las clases:
  - **Acoplamiento**
  - **Cohesión**

Acoplamiento

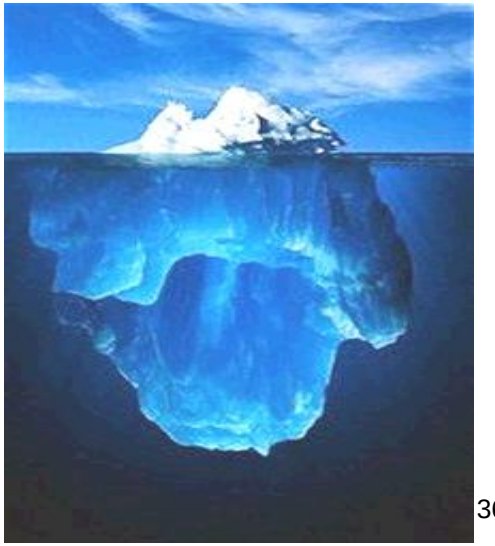


- Se refiere al grado de interconexión entre las clases de un programa.
  - Si dos clases dependen de muchos detalles de cada una, se dice que están fuertemente acopladas.
  - También se aplica a métodos y paquetes.
  - El grado de acoplamiento determina la dificultad de realizar cambios en una aplicación.
  - **Objetivo**: conseguir un acoplamiento débil o acoplamiento mínimo
- Conseguir clases cooperativas que se comuniquen a través de interfaces bien definidas.

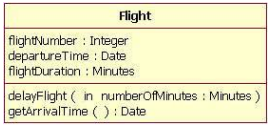
Acoplamiento



- Hay que reducir la interfaz de la clase al **mínimo**
  - La interfaz es un contrato que especifica un conjunto (público) de **métodos** y **propiedades** que estarán disponibles en cualquier objeto de una clase.
- No poner atributos públicos (excepto `final`). Si se usan públicos:
  - Hace que el código esté fuertemente acoplado.
  - Modificarlos implica cambios en los clientes.
  - Anula los beneficios de la encapsulación.
- Una clase (iceberg) mostrará solo lo que **puede hacer** pero nunca **como lo hace**.



Ejemplo (Word_of_Zuul)

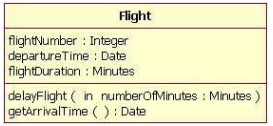



```
public class Room  {
    public String description;
    public Room northExit;
    public Room southExit;
    public Room eastExit;
    public Room westExit;

    public void setExits(Room north, Room east,
                         Room south, Room west){
        if(north != null)
            northExit = north;
        if(east != null)
            eastExit = east;
        if(south != null)
            southExit = south;
        if(west != null)
            westExit = west;
    }
}
```

- Queremos extender el código
- ¿Como se añaden nuevas salidas? → Es necesario modificar la clase!
- ¡Existe acoplameinto!

Como usar encapsulación



- Se puede usar `HashMap<String, Room>`
- Se pueden añadir nuevas salidas con lo que:
  - No se necesitan realizar cambios.
  - Se puede extender luego es mantenible.
  - **Es mantenible porque tiene un acoplamiento débil.**

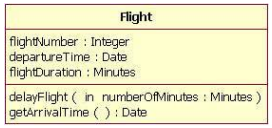
```
public class Room {
    private String description;
    private HashMap<String, Room> exits;

    public void setExits (String direction, Room room) {
        exists.put(direction, room);
    }
    public Room getExits (String direction) {
      return exists.get(direction);
    }
 }
```

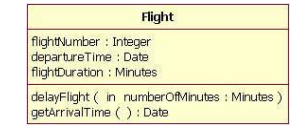
```
Room nerth = new Room();
Room r = new Room();
r.setExits("NE", nerth);
… = r.getExits("NE");
```

Objetivo: **acoplamiento mínimo**

Cohesión

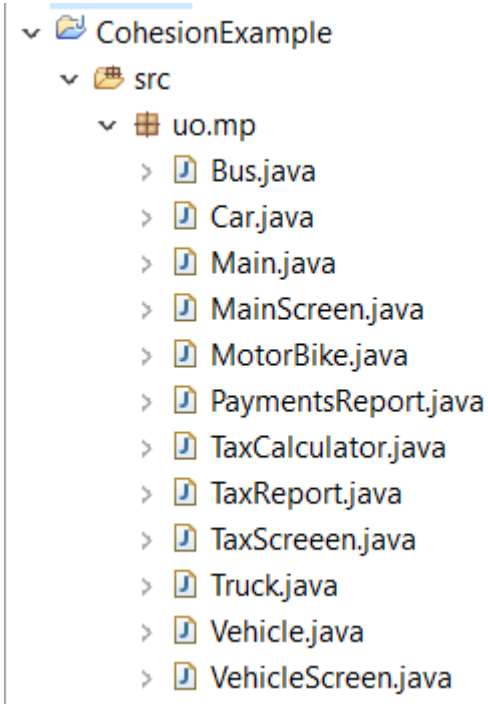


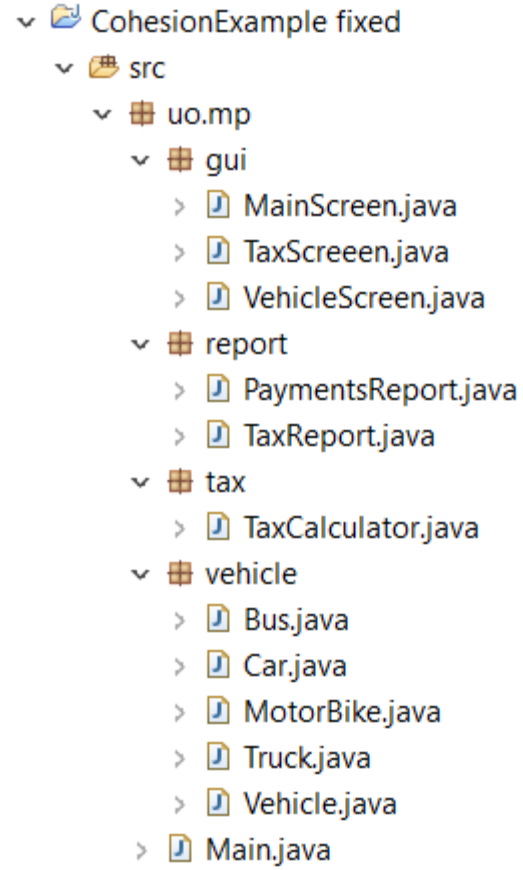
- Cohesión se refiere al **número** y **diversidad de tareas** de la que es responsable una clase.
- También se aplica a las métodos y paquetes.
- Cada **clase** debe representar **un** tipo de entidad
  - Entonces, se dice que tiene una cohesión alta.
  - Mejora el diseño en legibilidad y reutilización
- Cada método debe implementar **una** operación
- En un paquete, las clases deben estar relacionadas entre sí



Cohesion

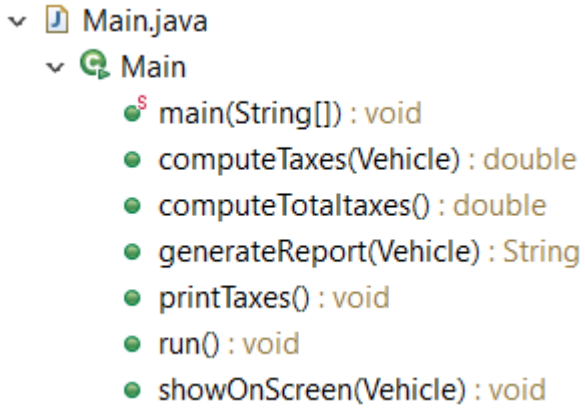
- ¿Qué distribución de paquetes es más cohesivo?

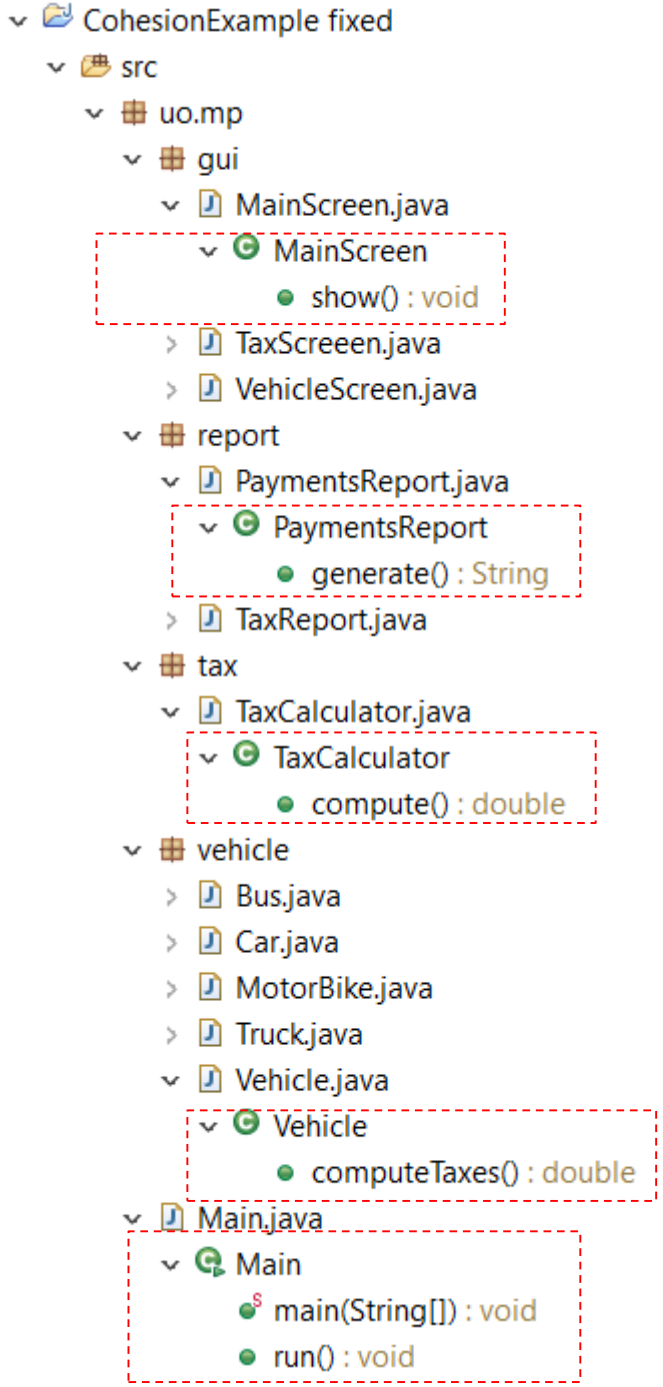




Cohesion

- ¿Qué diseño de clases es más cohesivo?





Cohesion

- ¿Qué método es más cohesivo?

A

```
public boolean isValid() {
    return isValidDay()
            && isValidMonth()
            && isValidDayInMonth();
}

private boolean isValidMonth() {
    return (month >= 1 && month <= 12);
}

private boolean isValidDay() {
    return (day >= 1 && day <= 31);
}

private boolean isValidDayInMonth() {
    return (this.day <= getMaxDaysInMonth());
}

private int getMaxDaysInMonth() {
    int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    return (month == 2 && isLeapYear()) ? 29 : days[ month - 1  ];
}

private boolean isLeapYear() {
    return (year % 400 == 0) || ((year % 4 == 0) && (year % 100 != 0));
}
```

```
public class Date {
    private int day;
    private int month;
    private int year;

    public Date(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }
```

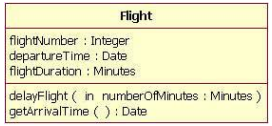
B

```
public boolean isValid() {
    if (day < 1 || day > 31) {
      return false;
    }
    if (month < 1 || month > 12) {
       return false;
    }

    // Days in the month
    int daysInMonth = 0;
    switch (month) {
      case 1:
      case 3:
      case 5:
      case 7:
      case 8:
      case 10:
      case 12: daysInMonth = 31;
               break;
      case 4:
      case 6:
      case 9:
      case 11 : daysInMonth = 30;
                break;
      case 2 :
      // Is leap
       if ( (year % 400 == 0) ||
           ( (year % 4 == 0) &&
             (year % 100 != 0) ) ) {
         daysInMonth = 29;
       }
       else {
           daysInMonth = 28;
       }
       break;
    }

    return (this.day <= daysInMonth);
}
```

Cohesión

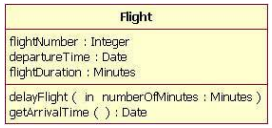



```
public class Game {
// … Se omite parte del código
 private void printWelcome() {
     System.out.println();
     System.out.println("Welcome to the World of Zuul!");
     System.out.println("Type 'help' if you need help.");
     System.out.println();
     System.out.println("You are"+
                     currentRoom.getDescription());
     System.out.print("Exits: ");
     if(currentRoom.northExit != null)
         System.out.print("north ");
     if(currentRoom.eastExit != null)
         System.out.print("east ");
     if(currentRoom.southExit != null)
         System.out.print("south ");
     if(currentRoom.westExit != null)
         System.out.print("west ");
     System.out.println();
 }  }
```

Mensaje de bienvenida

Mensaje de descripción de la habitación

Cohesión

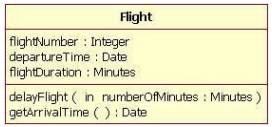



```
private void goRoom(Command command){
// … Se omite parte del código
String direction = command.getSecondWord();
Room nextRoom = currentRoom.getExit(direction);
  if (nextRoom == null) {
        System.out.println("There is no door!");
  }
  else {
    currentRoom = nextRoom;
    System.out.println("You are"
                + currentRoom.getDescription());
    System.out.print("Exits: ");
    if(currentRoom.northExit != null)
        System.out.print("north ");
    if(currentRoom.eastExit != null)
        System.out.print("east ");
    if(currentRoom.southExit != null)
        System.out.print("south ");
    if(currentRoom.westExit != null)
        System.out.print("west ");
    System.out.println(); }}
```

Ir a la habitación

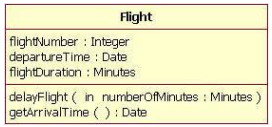
Mensaje de descripción de la habitación

Cohesión



- Los dos métodos anteriores implementan diferentes responsabilidades.
  - Pero existe **baja cohesión**
- El mensaje de descripción de la habitación esta **duplicado**.
- La duplicación del código es un indicador de mal diseño y hace que sea menos mantenible.
  - Los cambios deben realizarse en muchos sitios.
  - Las extensiones deben realizarse en muchos sitios
  - La depuración debe realizarse en muchos sitios
- **¡Eliminar el código duplicado!**
- **Objetivo**: **máxima cohesión**

Cohesión



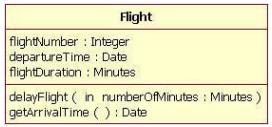

```
private void printLocationInfo() {
  System.out.println("You are“ +
                     currentRoom.getDescription());
  System.out.print("Exits: ");
  if(currentRoom.northExit != null)
        System.out.print("north ");
  if(currentRoom.eastExit != null)
        System.out.print("east ");
  if(currentRoom.southExit != null)
        System.out.print("south ");
  if(currentRoom.westExit != null)
        System.out.print("west ");
  System.out.println();

}
```

Mensaje de descripción de la habitación

Método cohesivo, corto y fácil de comprender. El nombre indica su propósito claramente

Cohesión




```
 private void goRoom(Command command) {
  // ...
 if (nextRoom == null) {
      System.out.println("There is no door!");
 }
 else {
    currentRoom = nextRoom;
    this.printLocationInfo();
 }
}
```

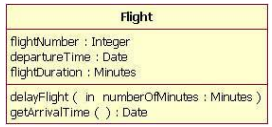
Ir a la habitación

Los cambios en `printLocationInfo()` se reflejan automáticamente

```
private void printWelcome() {
 System.out.println();
 System.out.println("Welcome to the World of Zuul!");
 System.out.println("Type 'help' if you need help.");
 System.out.println();
 this.printLocationInfo();
}
```

Mensaje de bienvenida

Diseño dirigido por responsabilidades

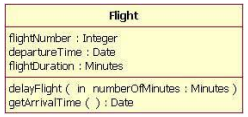


- **Diseño dirigido por responsabilidades (DDR)** es el proceso de diseñar clases asignado unas responsabilidades bien definidas a cada clase.

  Expresa la idea de que cada clase debe ser responsable de gestionar sus propios datos

- Es otro principio de diseño para obtener una **cohesión alta.**
- Si se necesita añadir nueva funcionalidad a la aplicación. ¿Qué clase es responsable?
  - Colocar los nuevos métodos en aquellas clases que proporcionen los datos que requieren.

La clase responsable de almacenar unos datos determinados, debe ser responsable de manipularlos a través de los métodos.



Resumiendo: cohesión y acoplamiento

- En un buen diseño hay **máxima cohesión**
  - Los **métodos** deben implementar una sola tarea
  - Una **clase** debe representar una única entidad
  - Las **clases** de un **paquete** debe estar relacionadas
- En un buen diseño hay **mínimo acoplamiento**
  - Las **clases** deben reducir sus interfaces el mínimo
  - No pasar demasiados parámetros a los **métodos**
  - Reducir el número de clases públicas en un **paquete**

Refactorización

> "**Refactorización** es el proceso de **reestructuración** de **clases** y **métodos** existentes para mejorar características no funcionales del software

- En la vida de una aplicación, la funcionalidad se añade gradualmente
  - El software evoluciona, cambia, es mutable ...
- Refactorización es **volver a pensar** y **rediseñar** la estructura de las clases y métodos.
  - Las clases pueden ser divididas en dos
  - Los métodos pueden ser divididos en dos o más
  - El objetivo es: **máxima cohesión y mínimo acoplamiento**

Ejemplo de Refactorización


```
public class Game {
 private void printWelcome() {
     System.out.println();
     System.out.println("Welcome to the World of Zuul!");
     System.out.println("Type 'help' if you need help.");
     System.out.println();
     System.out.println("You are"+
                        currentRoom.getDescription());
     System.out.print("Exits: ");
     if(currentRoom.northExit != null)
         System.out.print("north ");
     if(currentRoom.eastExit != null)
         System.out.print("east ");
     if(currentRoom.southExit != null)
         System.out.print("south ");
     if(currentRoom.westExit != null)
         System.out.print("west ");
     System.out.println();
 }
}
```

Mensaje de bienvenida

Mensaje
de descripción
de la habitación

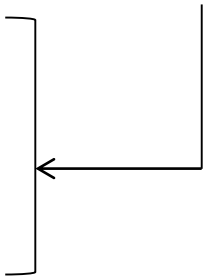
Refactorización

Extract Method

Ejemplo de Refactorización


```
public class Game {
 private void printWelcome() {
     System.out.println();
     System.out.println("Welcome to the World of Zuul!");
     System.out.println("Type 'help' if you need help.");
     System.out.println();
     this.printLocationInfo();
 }
private void printLocationInfo() {
   System.out.println("You are"  +
                      currentRoom.getDescription());
   System.out.print("Exits: ");
   if(currentRoom.northExit != null)
         System.out.print("north ");
   if(currentRoom.eastExit != null)
         System.out.print("east ");
   if(currentRoom.southExit != null)
         System.out.print("south ");
   if(currentRoom.westExit != null)
         System.out.print("west ");
   System.out.println();
```

Mensaje de bienvenida



- Esta refactorización se denomina ***Extract Method***
- Está soportado en Eclipse

Ejercicio de refactorización


```
public class Date {

 private int day;
 private int month;
 private int year;

    public Date(int day, int month, int year)
    {
      this.day = dia;
      this.month = month;
      this.year = year;
    }
```

Ejercicio de refactorización

Refactoriza el siguiente código

```
public boolean isValid ()
{
  if (day < 1 || day > 31)
      return false;
  if (month < 1 || month > 12)
      return false;

// Día del mes
 int daysInMonth = 0;
 switch (month) {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12: daysInMonth = 31;
           break;
case 4:
case 6:
case 9:
case 11 : daysInMonth = 30;
          break;
case 2 :
// verificación de año bisiesto
  if ( (year % 400 == 0) ||
     ( (year % 4 == 0) &&
       (year % 100 != 0) ) )
         daysInMonth = 29;
  else
       daysInMonth = 28;
       break;
  }
  return (day <= daysInMonth)

 } // Del método isValid
} //De la clase Date
```

Ejercicio de refactorización


```
private boolean isLeap() {
   return (year % 400 == 0) ||
          ( (year % 4 == 0) &&
            (year % 100 != 0) );
 }


private int daysInMonth() {
   switch (month) {
     case 1: case 3: case 5: case 7: case 8: case 10: case 12:
             return 31;
     case 4: case 6: case 9: case 11 :
             return 30;
     case 2: return isLeap()?29:28;
   }
   return -1;
 }
```

Ejercicio de refactorización


```
public boolean isValid() {
  if (day < 1 || day > 31)
    return false;
  if (month < 1 || month > 12)
    return false;
  return this.day <= daysInMonth();
}
```

Un buen libro que describe esta técnica es:
"Refactoring: Improving the Design of Existing Code" by **Martin Fowler**

Refactorización y pruebas

- Cuando se modifica algo existe la posibilidad de que se introduzcan errores.
- **La refactorización debe seguir dos pasos**:
  - Debemos asegurarnos de que exista un conjunto de tests correctos para la versión actual de la aplicación.
  - Reestructurar el código para mejorar su estructura, no para cambiar o aumentar su funcionalidad.

    ⟹ Ejecutar las pruebas regresivas
- Cuando haya finalizado la refactorización, se puede cambiar la aplicación (mejorar la funcionalidad)

  ⟹ Ejecutar las pruebas regresivas
- Usar herramientas para pruebas automáticas (JUnit)

Refactorización para la independencia de idioma. **Tipos enumerados**

- Siguiendo con el ejemplo del juego Word_of_Zuul, la interfaz del usuario está ligada a comandos escritos en inglés (clases **`ComandWords`** y **`Game`**).
- Para conseguir que sea independiente del idioma, hay que tener un único lugar para almacenar el texto real de las palabras de los comandos.
- Una característica del lenguaje que permite esto son los **`tipos enumerados`** o **`enums`**.

```
public enum CommandWord
{ //un valor para cada palabra de commando
   GO, QUIT, HELP, UNKNOW
}
```

Por convenio, se escriben en mayúsculas

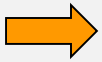
Cada nombre es una instancia diferente de ese tipo

Tipos enumerados

- Ventajas del uso de tipos enumerados:
  - Alta legibilidad
  - Comprobación de tipos estático
  - Mejor rendimiento en ejecución que los Strings

Uso del tipo enumerado en el método **processCommand** de **Game**

```
private boolean processCommand(Command command)
{
 boolean wantToQuit = false;
 String word = command.getCommandWord();
 if (word.equals("help")) {
    printHelp();
 } else if (word.equals("go")) {
    goRoom(command);
 } else if (word.equals("quit")) {
    wantToQuit = quit(command);
 }
```



```
CommandWord word = command.getCommandWord();
if (word == CommandWord.HELP) {
    printHelp();
} else if (word == CommandWord.GO) {
    goRoom(command);
} else if (word == CommandWord.QUIT)) {
    wantToQuit = quit(command);
}
```

…

Uso de tipos enumerados

- Los comandos escritos por el usuario se pueden asociar a los valores correspondientes del tipo enumerado.

Uso del tipo enumerado en la clase **CommandWords**

Tipo enumerado

```
public class CommandWords
{ private HashMap<String, CommandWord> validCommands;

 public CommandWords()
 {
   validCommands = new HashMap<>();
   validCommands.put ("go", CommandWord.GO);
   validCommands.put ("help", CommandWord.HELP);
   validCommands.put ("quit", CommandWord.QUIT);
 }
  …
}
```

Revisión

- El software está cambiando continuamente.
- **Objetivo**: conseguir software de calidad (comprensible, fácil de mantener...)
- El acoplamiento y la cohesión son las bases de un buen diseño del software
  - alta cohesión y bajo acoplamiento
- El estilo de codificación (comentarios, nombres, presentación, etc) también es importante.
- Refactorización y pruebas son dos herramientas importantes para construir software de calidad
- Refactorización y pruebas se aplican durante el tiempo de vida de un programa: *implementar, probar, refactorizar, probar la refactorización, implementar, probar, etc.*