



# Diseñando clases

Cómo escribir clases de tal manera  
que sean fácilmente entendibles,  
mantenibles y reusables

# Principales conceptos a ser cubiertos

- Diseño dirigido por la responsabilidad (RDD)
- Acoplamiento
- Cohesión
- Refactorización

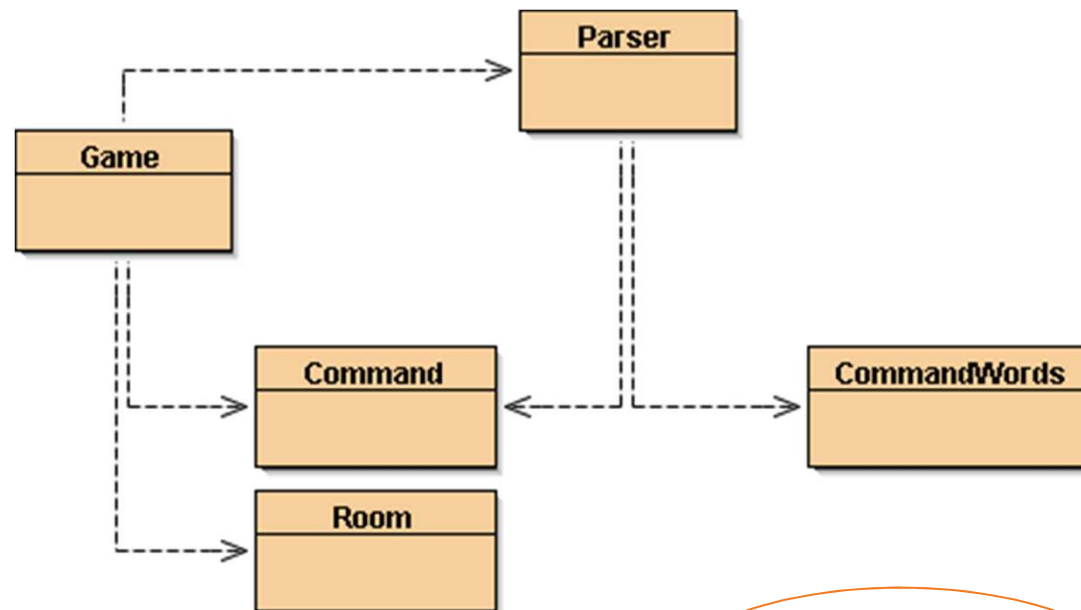
# Cambios software

- El software no es como una novela que *es escrita una vez* y se mantiene intacta.
- El software es extendido, corregido, mantenido, portado, adaptado, ...
- El trabajo se hace por gente diferente a lo largo del tiempo (a menudo décadas).

# Cambie o muera

- Solo hay dos opciones para software:
  - O es continuamente mantenido,
  - o muere.
- El software que no puede ser mantenido será desechado.

# Mundo de Zuul

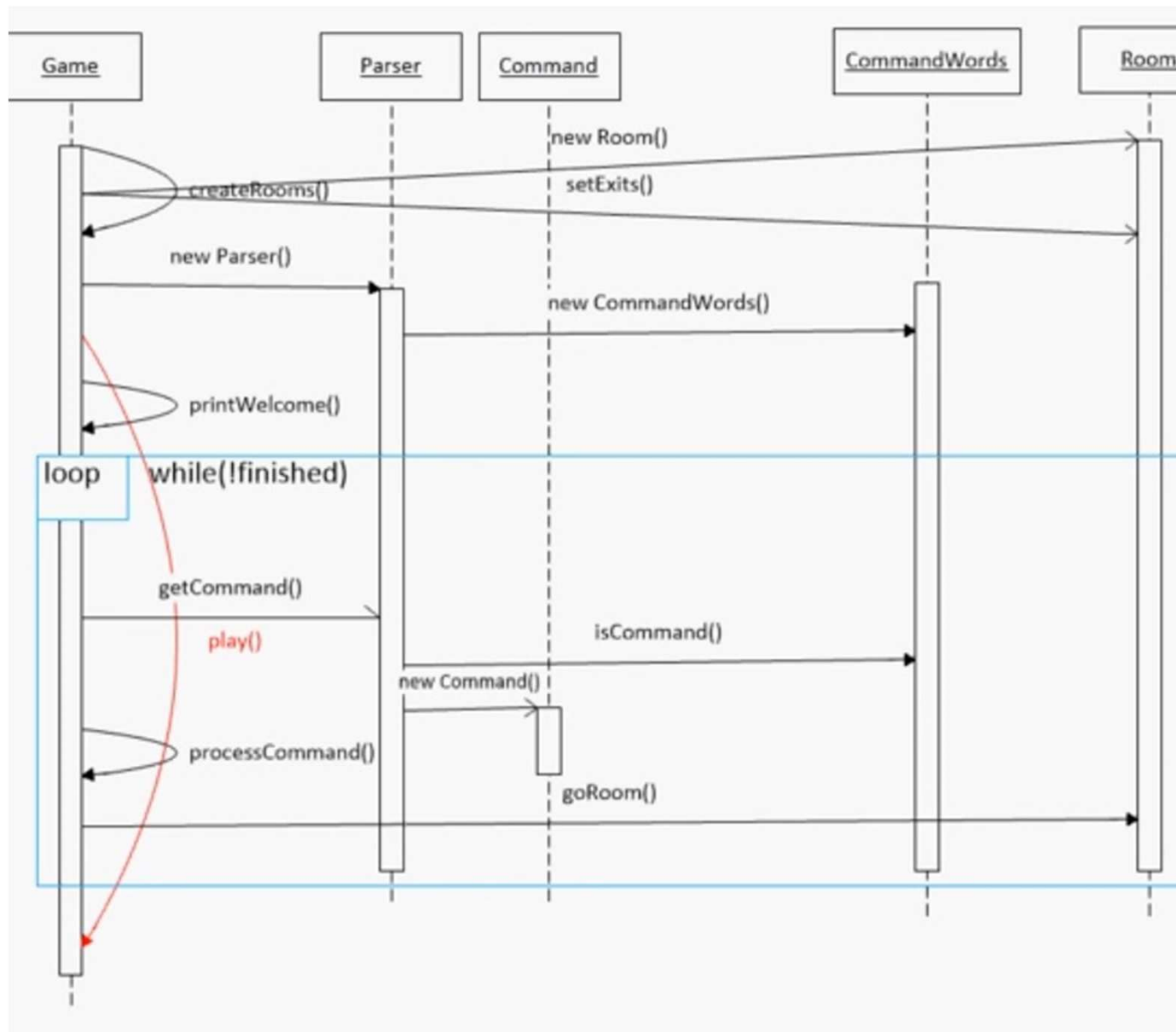


Explore  
zuul-bad

# Las clases de Zuul

- **Game:** El punto de partida y el ciclo de control principal.
- **Room:** Una habitación en el juego.
- **Parser:** Lee las entradas del usuario.
- **Command:** Un comando del usuario.
- **CommandWords:** Comandos de usuario reconocidos.





# Calidad del código y del diseño

- Si vamos a ser críticos sobre la calidad del código, necesitamos un criterio de evaluación.
- Dos conceptos importantes para abordar la calidad del código son:
  - Acoplamiento
  - Cohesión



# Acoplamiento

- Acoplamiento se refiere a vínculos entre unidades separadas de un programa.
- Si dos clases dependen cercanamente en muchos aspectos una de la otra, decimos que están *fuertemente* emparejadas.
- Nuestro objetivo es un acoplamiento *débil*.
- Un diagrama de clase provee (limitados) indicios del grado de acoplamiento.

# Cohesión

- Cohesión se refiere al número y diversidad de tareas de las cuales una sola unidad es responsable.
- Si cada unidad es responsable de una sola tarea lógica, decimos que tiene *alta* cohesión.
- El objetivo es una cohesión alta.
- ‘Unidad’ aplica a clases, métodos y módulos (paquetes).


# Un ejemplo para probar calidad

- Agregue dos nuevas direcciones a ‘Mundo de Zuul’:
  - “arriba”
  - “abajo”
- ¿Qué necesita cambiar para hacer esto?
- ¿Que tan fáciles son los cambios para aplicarlo a fondo?



# Diseñando Clases

Acoplamiento, cohesión y diseño  
impulsado por la responsabilidad.



# Acoplamiento (repetición)

- Acoplamiento se refiere a vínculos entre unidades separadas de un programa.
- Si dos clases dependen cercanamente en muchos aspectos una de la otra, decimos que están *fuertemente* acopladas.
- Nuestro objetivo es un acoplamiento *débil*.
- Un diagrama de clase provee indicios (limitados) del grado de acoplamiento.

# Acoplamiento débil

- Apuntamos hacia acoplamiento débil
- El acoplamiento débil hace posible:
  - entender una clase sin leer otras;
  - cambiar una clase con poco o ningún efecto sobre otra clase
- Así: el acoplamiento débil incrementa la mantenibilidad.



# Acoplamiento fuerte

- Intentamos evitar el acoplamiento fuerte.
- Cambios en una clase conllevan a cambios en cascada a otras clases.
- Clases son más difíciles de entender en aislamiento.
- Flujo de control entre objetos de diferentes clases es complejo.

# Cohesión (redundancia)

- Cohesión se refiere al número y diversidad de tareas de las cuales una sola unidad es responsable.
- Si cada unidad es responsable de una sola tarea lógica, decimos que tiene *alta* cohesión.
- El objetivo es una cohesión alta.
- ‘Unidad’ aplica a clases, métodos y módulos (paquetes).

# Alta cohesión

- Apuntamos hacia la alta cohesión.
- Alta cohesión hace más fácil:
  - entender lo que una clase o método hace;
  - usar nombres descriptivos para variables, métodos y clases;
  - reutilizar clases y métodos.

# Baja cohesión

- Nuestro objetivo es evitar clases y métodos poco cohesionados.
  - Métodos que realizan múltiples tareas.
  - Clases no tienen identidad clara.

# Consejos sobre cohesión

- Métodos
  - Mantenga los métodos cortos y ‘al grano’.
  - Métodos complejos son difíciles de entender, cuando se revisan más tarde.

# Cohesión aplicada en diferentes niveles

- Nivel de clase:
  - Las clases deberían representar una única entidad bien definida.
- Method level:
  - Un método debería ser responsable de una única tarea bien definida.



# Duplicación de código

- Duplicación de código
  - es un indicador de mal diseño,
  - hace más difícil el mantenimiento,
  - puede llevar a introducción de errores durante el mantenimiento.

# Diseño impulsado por la responsabilidad

- Pregunta: ¿dónde deberíamos agregar un nuevo método (qué clase)?
- Cada clase debe ser responsable de manipular sus propios datos.
- La clase que posee los datos debe ser responsable de procesarla.
- RDD conduce a acoplamiento débil.

# Localizando el cambio

- Uno de los objetivos de reducir el acoplamiento y el diseño basado en la responsabilidad es localizar el cambio.
- Cuando se necesita un cambio, se deben afectar tan pocas clases como sean posibles.

# Pensando por adelantado

- Al diseñar una clase, tratamos de pensar qué cambios es probable que se hagan en el futuro.
- Nuestro objetivo es facilitar esos cambios.

# Refactorización

- Cuando se hace mantenimiento a las clases, a menudo se agrega código.
- Las clases y métodos tienden a ser más largos.
- De vez en cuando, las clases y métodos deben ser refactorizados para mantener la cohesión y el bajo acoplamiento.

# Refactorización y pruebas

- Cuando refactorice el código, separe la refactorización de otros cambios.
- Primero haga la refactorización solamente, sin cambiar la funcionalidad.
- Pruebe antes y después de refactorizar para asegurarse de que no haya roto nada (¿ó: “no haya roto nada”?).



# Preguntas de diseño

- Preguntas comunes:
  - ¿Cuán larga debería ser una clase?
  - ¿Cuán largo debería ser un método?
- Estas pueden ser ahora respondidas en términos de cohesión y acoplamiento.

# Guía de diseño

- Un método es demasiado largo si realiza más de una tarea lógica.
- Una clase es demasiado compleja si representa más de una entidad lógica.
- Nota: estas son *pautas*, aún dejan mucho espacio abierto para el diseñador.

# Tipos Enumerados

- Una característica de lenguaje.
- Use `enum` en vez de `class` para introducir un nombre de tipo.
- Su uso más simple es definir un conjunto de nombres significativos.
  - Alternativa para constantes `int` estáticas.
  - Cuando los valores de las constantes serían arbitrarios.

# Un tipo enumerado básico

```
public enum CommandWord
{
    // A value for each command word,
    // plus one for unrecognised commands.
    GO, QUIT, HELP, UNKNOWN;
}
```

- Cada nombre representa un objeto de tipo enum, p. ej., `CommandWord.HELP`.
- Los objetos enum no son creados directamente.
- Las definiciones enum también pueden tener campos, constructores y métodos.

# Resumen

- Los programas son cambiados continuamente.
- Es importante hacer estos cambios posibles.
- La calidad del código requiere mucho más que solo hacerlo correcto una vez.
- El código debe ser entendible y mantenible.

# Resumen

- El código de buena calidad evita la duplicación, muestra alta cohesión, bajo acoplamiento.
- El estilo de codificación (comentarios, nombres, diseño, etc.) también es importante.
- Existe una gran diferencia en la cantidad de trabajo requerido para cambiar el código mal estructurado y bien estructurado.