Diseño de clases

Cómo escribir clases en una forma que sean fácilmente entendibles, mantenibles y reusables.

Principales conceptos que se abordan en esta unidad

- Diseño dirigido por responsabilidades
- Acoplamiento
- Cohesión
- Refactorización

Cambios en el software

- El software no es como una novela que se escribe una vez y permanece sin cambios.
- El software es extendido, corregido, mantenido, portado, adaptado…
- El trabajo se realiza por diferentes personas a lo largo del tiempo (a menudo, durante décadas).

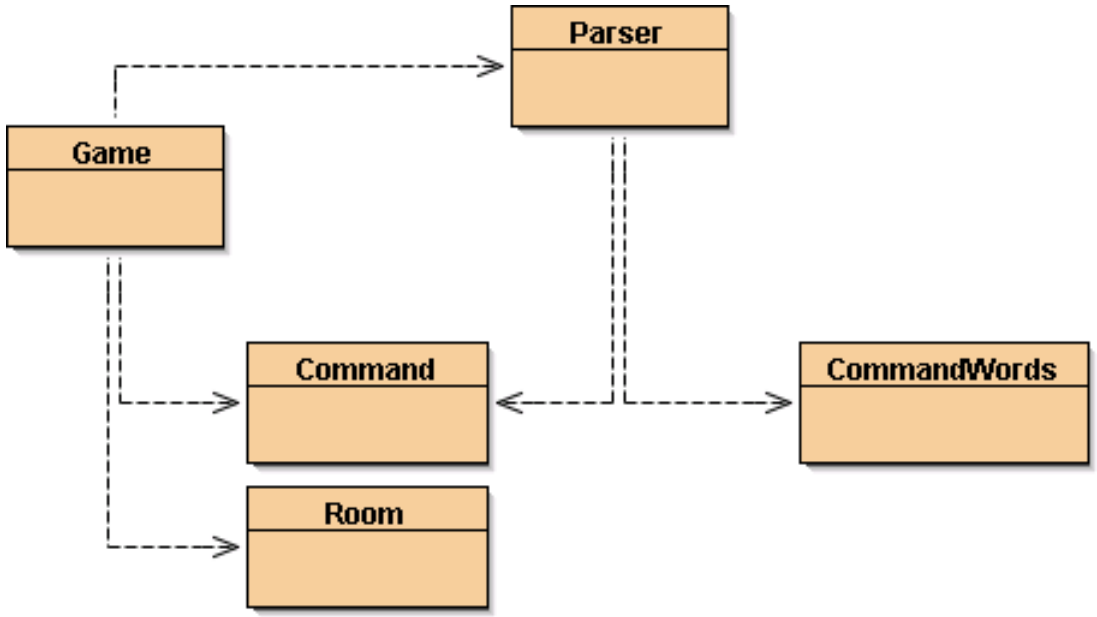
  Ej.: 7.1, 7.2

Cambiar o morir

- Solo hay dos opciones para el software:
  - Es contínuamente mantenido
  - O muere.
- El software que no pueda ser mantenido será descartado.

Ej.: 7.3

El mundo de Zuul



Calidad del código

- Dos conceptos importantes para la calidad del código:
  - Acoplamiento
  - Cohesión

Ej.: 7.4

Duplicación de código

- Es un indicador de un mal diseño
- Hace más arduo el mantenimiento
- Puede llevar a la introducción de errores durante el mantenimiento.

Ej.: 7.5

Acoplamiento

- Se refiere a los enlaces entre unidades separadas de un programa.
- Si dos clases dependen una de la otra en muchos detalles, se dice que estan *fuertemente acopladas*.
- Se tiende al *acoplamiento débil*.

Ej.: 7.6, 7.7

Acoplamiento débil

- El acoplamiento débil hace posible:
  - Entender una clase sin leer otras.
  - Cambiar una clase sin afectar las otras.
  - Así, se mejora el mantenimiento.

Ej.: 7.8

Diseño dirigido por responsabilidad

- Pregunta: ¿dónde se debería añadir un nuevo método (en cuál clase)?
- Cada clase debe ser responsable de la manipulación de sus datos.
- La clase que posee los datos debe ser responsable de procesarlos.
- El Diseño Dirigido por Responsabilidad (DDR) lleva al bajo acoplamiento.

Ej.: 7.9, 7.10, 7.11, 7.12, 7.13

Localizando el cambio

- Uno de los objetivos de reducir el acoplamiento y del diseño dirigido por responsabilidad es localizar el cambio.

Pensar en futuro

- Cuando se diseña una clase, se trata de pensar cuales cambios posiblemente se harán en el futuro.
- Se trata de facilitar esos cambios.

Ej.: 7.14, 7.15, 7.16, 7.17, 7.18, 7.19

La cohesión

- Se refiere al número y diversidad de tareas de las que una unidad es responsable.
- Si cada unidad es responsable de una sola tarea lógica, se dice que tiene *cohesión alta*.
- El concepto de cohesión se aplica a las clases y los métodos.
- Se tiende a la *cohesión alta*.

Cohesión alta

- La cohesión alta hace más fácil:
  - Entender lo que hace una clase o método,
  - Usar nombres descriptivos,
  - Reusar clases o métodos.

Cohesión de clases

- Las clases deberían representar una entidad bien definida.

Ej.: 7.20, 7.21, 7.22

Cohesión de métodos

- Un método debería ser responsable de una única tarea bien definida.

Ej.: 7.23, 7.24, 7.25, 7.26

Refactorización

- Cuando se mantienen las clases, a menudo se añade código.
- Las clases y los métodos tienden a hacerse más grandes.
- Periódicamente, las clases y los métodos deberían *refactorizarse* para mantener la cohesión y el bajo acoplamiento.

Refactorización y prueba

- Cuando se refactoriza código, se debe separar esta tarea de la realización de otros cambios.
- Primero hacer solo la refactorización, sin cambiar la funcionalidad.
- Realizar pruebas antes y después de la refactorización para asegurar que nada se rompió.

Ej.: 7.27, 7.28, 7.29, 7.30, 7.31, 7.32, 7.33, 7.34

Tipos enumerados

- Es una propiedad del lenguage.
- Se usa `enum` en vez de `class` para introducir un nombre de tipo.
- El uso más simple es definir un conjunto significativo de nombres.
  - Alternativa a constantes `static int`

Ej.: 7.35, 7.36, 7.37, 7.38, 7.39, 7.40, 7.41

Un tipo enumerado básico

```
public enum PalabraComando
{
    // Un valor para cada palabra comando,
    // más una para comandos irreconocibles.
    IR, SALIR, AYUDA, DESCONOCIDA;
}
```

- Cada nombre representa un objeto del tipo enumerado, p.ej. **`PalabraComando.AYUDA`**.
- Los objetos enumerados no se crean directamente por el programador.

Preguntas sobre el diseño

- Preguntas comunes como:
  - ¿Cuán grande debe ser una clase?
  - ¿Cuán grande debe ser un método?
- Ahora pueden responderse en términos de la cohesión y el acoplamiento.

Pautas de diseño

- Un método es demasiado grande si hace más de una tarea lógica.
- Una clase es demasiado compleja si representa más de una entidad lógica.
- **Nota**: éstas sólo son *pautas* – todavía dejan mucho espacio abierto al diseñador.

Ej.: 7.42, 7.43, 7.44, 7.45, 7.46, 7.47, 7.48, 7.49

Ejecutar un programa fuera de BlueJ

- Se requiere que el juego desarrollado pueda pasarse a otras personas para que lo jueguen sin necesidad de abrir un editor de Bluej.
- Métodos de clase o estáticos.
- El método main.

Ej.: 7.50, 7.51, 7.52, 7.53, 7.54

Resumen

- Los programas continuamente deben cambiarse.
- Es importante hacer posible este cambio.
- La calidad del código requiere mucho más que ejecutar correctamente una vez.
- El código debe ser comprensible y mantenible.
- Códigos de buena calidad evitan la duplicación, muestran alta cohesión y bajo acoplamiento.

Resumen

- El estilo de codificación (comentarios, nombres, disposición, etc.) también es importante.
- Hay una gran diferencia en la cantidad de trabajo requerida para cambiar código pobremente estructurado y otro bien estructurado.

Ej.: 7.55, 7.56, 7.57