Fundamentos de la Programación Orientada a Objetos

Diseño de clases

Programación Orientada a Objetos
Facultad de Informática

Juan Pavón Mestras
Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense Madrid

Los cambios en el software

- El software cambia
  - Los requisitos cambian
  - Se necesita nueva funcionalidad
  - Mejoras de eficiencia
  - Corrección de errores
  - Adaptación a plataformas, componentes
- Y a lo largo de su vida pasa por muchas manos
  - Múltiples desarrolladores
- El software que no se mantiene cae en desuso
  - ¿Qué software hay en tu PC que no hayas actualizado alguna vez?

Diseño de clases

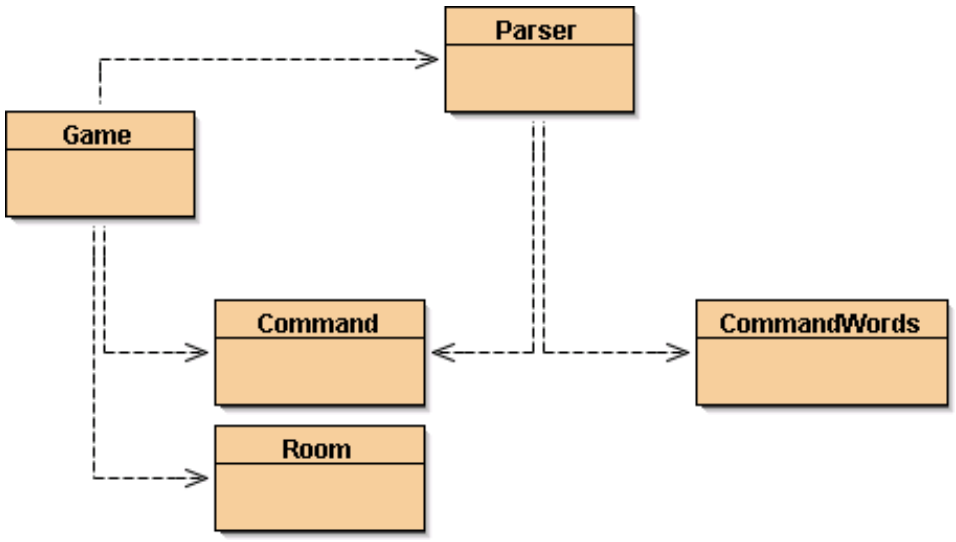
- Características de un buen programa
  - Fácil de entender
  - Mantenible
  - Reutilizable
- Hay principios de diseño que ayudan en el diseño de las clases y la estructura del programa:
  - Diseño dirigido por responsabilidades
  - Acoplamiento y cohesión
  - Refactorización

Ejemplo de trabajo: El mundo de Zuul

- Juego de aventuras
  - Veremos cómo ir mejorando el diseño inicial
- *Colossal Cave Adventure* (Will Crowder, ’70)
  - Juego que trata de encontrar un camino a través de un complejo sistema de cuevas, ubicar el tesoro escondido, usar palabras secretas y otros misterios, todo ello con el propósito de obtener el máximo número de puntos

Ejemplo de trabajo: El mundo de Zuul

- Ejemplo de código no muy ejemplar
  - chapter07/zuul-bad



El mundo de Zuul

- Clase Game
  - Clase principal que inicia el juego y permite empezar un ciclo de lectura y ejecución de comandos
  - También tiene el código que implementa cada comando
- Clase CommandWords
  - Define todas las palabras de posibles comandos
- Clase Command
  - Representa un comando introducido por el usuario y permite controlar si es válido y considerar por separado la primera y segunda palabras
- Clase Room
  - Representa una habitación que puede tener varias salidas para ir a otras habitaciones
- Clase Parser
  - Se encarga de interpretar la entrada del usuario y trata de crear los objetos Command correspondientes

Calidad del diseño de clases

- Acoplamiento
  - Interconectividad de las clases
    - Si dos clases dependen mutuamente en muchos detalles se dice que están fuertemente acopladas
  - Una buena propiedad es el acoplamiento débil
    - Que las clases sean lo más independientes posibles
    - Y que se comuniquen a través de pequeñas interfaces bien definidas
- Cohesión
  - Cuánto se ajusta una unidad de código a una tarea lógica o a una entidad
    - El número y diversidad de tareas que se asignan a una unidad
  - Una buena propiedad es una alto grado de cohesión
    - Cada unidad de código (método, clase o módulo) es responsable de una tarea bien definida o de una entidad

Calidad del diseño de clases

- Un sistema fuertemente acoplado (esto es, con muchas dependencias entre clases)
  - ¿Será más o menos fácil de modificar?
  - ¿Afectarán sus cambios a otras clases?
  - ¿Mejorará su mantenibilidad?
- Si un método es responsable de una única cosa bien definida seguramente podrá ser usado nuevamente en un contexto diferente
  - Más fácil entender lo que hace
  - Asignar nombres más descriptivos
  - Mayor reutilización

Calidad del diseño de clases

- Cohesión de métodos
  - Cada método debe ser responsable de una y solo una tarea bien definida
- Cohesión de clases
  - Cada clase debe representar una sola entidad bien definida

Duplicación del código

- Un indicador de mala calidad de diseño
  - Hace más difícil el mantenimiento del código
  - Puede inducir a la introducción de errores durante el mantenimiento
    - Inconsistencias
    - Se cambia el código en un sitio pero no en otro similar
    - Ejemplo: métodos *printWelcome()* y *goRoom()* en la clase *Game*
      - Ambos métodos imprimen información pero ninguno puede llamar al otro porque cada uno de ellos, además, hace otras cosas
    - ¿La solución?
      - Refactorizar código en un método:
        private void printLocation()

Hacer extensiones

- Supóngase que se quiere añadir la posibilidad de ir en otras direcciones, por ejemplo: *go up* y *go down*
  - *¿Qué clases y métodos habrá que tocar?*
    - *Room*
    - *Game*
    - *Están fuertemente acopladas*
- Una solución al acoplamiento fuerte: el **encapsulamiento**

Hacer extensiones

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

// utilización:

```
 // Try to leave current room.
        Room nextRoom = null;
        if(direction.equals("north")) {
            nextRoom = currentRoom.northExit;
        }
        if(direction.equals("east")) {
            nextRoom = currentRoom.eastExit;
        }
        if(direction.equals("south")) {
            nextRoom = currentRoom.southExit;
        }
        if(direction.equals("west")) {
            nextRoom = currentRoom.westExit;
        }
```

Hacer extensiones

```
public class Room // encapsulando
{
   private String description;
   private Room northExit;
   private Room southExit;
   private Room eastExit;
   private Room westExit;
…
   public Room getExit(String direction) {
        if (direction.equals(“north”)) return northExit;
        if (direction.equals(“south”)) return southExit;
        if (direction.equals(“east”)) return eastExit;
        if (direction.equals(“west”)) return westExit;
   }
}
```

```
// utilización:
 // Try to leave current room.
        Room nextRoom = currentRoom.getExit(direction);
```


Hacer extensiones

- Prueba a mejorar la manera de imprimir la información de ubicación (método printLocation())
  - Esto será más fácil con un método que describa las salidas disponibles de una habitación en la clase Room:

    public String getExitString()

Hacer extensiones

- Lo mejor de todo es que ahora se puede utilizar cualquier representación interna para las salidas de una habitación ya que ninguna clase accede directamente a esos campos
- Por ejemplo, usando un *HashMap*
  - *Ver código en chapter07/zuul-better*
  - La clase es más corta
  - Métodos más simplificados
  - Más fácil de ampliar los tipos y el número de salidas

Diseño dirigido por responsabilidades

- Asignar responsabilidades bien definidas a cada clase
  - ¿En qué clase habrá que poner un nuevo método?
  - Cada clase es responsable de gestionar sus propios datos
  - La clase que posee unos datos tiene que ser responsable de procesarlos
- Un buen diseño dirigido por responsabilidades conduce a reducir el grado de acoplamiento

Localización de cambios

- Uno de los objetivos de un buen diseño de clases es facilitar la localización de los cambios
  - Las modificaciones en una clase deberían tener efectos mínimos sobre las otras clases
  - Por eso hay que evitar definir campos públicos y definir claramente la interfaz de uso de la clase como un conjunto de métodos públicos
  - Los métodos públicos de la clase no deberían cambiar
    - Si acaso añadir otros nuevos

Acoplamiento implícito

- Cuando una clase depende de la información interna de otra pero esta dependencia no es inmediatamente obvia
  - El uso de campos públicos puede ser obvio
    - Si se cambian se producirán errores al compilar las clases que los usen
  - Pero hay casos más difíciles de detectar
    - Ejemplo: ¿qué pasa si se quieren añadir más comandos en el juego?
    - Ejercicio: Agrega el comando “eat” para que cuando se ejecute aparezca el texto “I have eaten and I am not more hungry”

Pensar en futuro

- Al diseñar clases hay que pensar cómo podrían ampliarse en el futuro
  - ¿Qué podrá cambiar?
- La encapsulación ayuda mucho

Refactorización

- Al ir modificando las clases normalmente se va añadiendo más y más código
  - Las clases y los métodos suelen crecer
  - Habrá que refactorizar para mantener buenos niveles de cohesión y bajo acoplamiento
- La refactorización no se tiene que hacer a la vez que otros cambios
  - Primero hacer la refactorización sin cambiar la funcionalidad
  - Luego probar el código resultante para asegurarse que sigue funcionando correctamente

Cuestiones de diseño

- ¿Qué tamaño debe tener una clase?
- ¿Qué tamaño debe tener un método?
  - La respuesta está en sus responsabilidades, la cohesión y el acoplamiento

Cuestiones de diseño

- Un método es demasiado largo si hace más de una tarea lógica
- Una clase es demasiado compleja si representa más de una entidad lógica

Como ejercicio

- Más extensiones al juego
  - Cambiar el idioma de los comandos (por ejemplo, a español)
  - Hacer un programa main() que permita ejecutar el juego fuera de BlueJ
  - Definir una configuración de habitaciones en tres dimensiones para el juego y probarla
  - Añadir llaves en el juego que puedan estar en habitaciones y que sirvan para abrir puertas especiales
  - Añadir puntuación al juego
  - Invéntate una extensión…