Designing classes

How to write classes in a way that they are easily understandable, maintainable and reusable

Chapter-06;
Objects First with Java using BlueJ

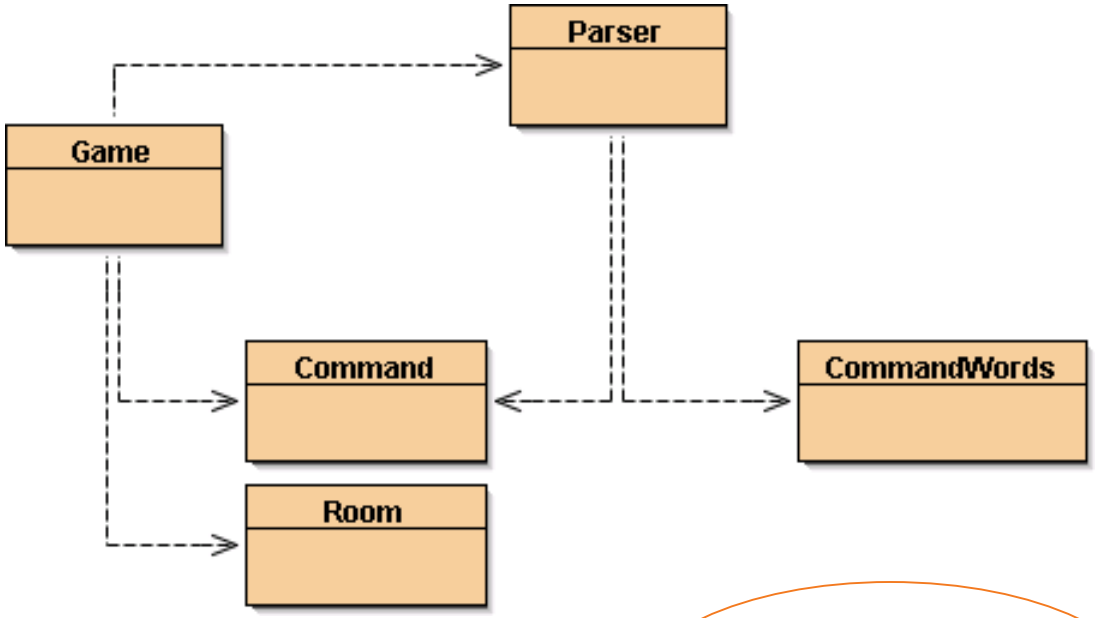
Software changes

- Software is not like a novel that is written once and then remains unchanged.
- Software is extended, corrected, maintained, ported, adapted, …
- The work is done by different people over time (often decades).

Change or Die

- There are only two options for software:
  - either it is continuously maintained
  - or it dies!
- Software that cannot be maintained will be thrown away.

World of Zuul



Explore
`zuul-bad`

The Zuul Classes

- **`Game`**: The starting point and main control loop.
- **Room**: A room in the game.
- **Parser**: Reads user input.
- **Command**: A user command.
- **CommandWords**: Recognized user commands.

Code and Design Quality

- If we are to be critical of code quality, we need evaluation criteria.
- Two important concepts for assessing the quality of code are:
  - Coupling
  - Cohesion

Coupling

- Coupling refers to links between separate units of a program.
- If two classes depend closely on many details of each other, we say they are *tightly coupled*.
- *We aim for loose coupling.*
- A class diagram provides (limited) hints at the degree of coupling.

Loose Coupling

- We aim for loose coupling.
- Loose coupling makes it possible to:
  - understand one class without reading others;
  - change one class with little or no effect on other classes.
- Thus:
  loose coupling increases maintainability.

Tight Coupling

- We try to avoid tight coupling.
- Changes to one class bring a cascade of changes to other classes.
- Classes are harder to understand in isolation.
- Flow of control between objects of different classes is complex.

Cohesion

- Cohesion refers to the number and diversity of tasks that a single unit is responsible for.
- If each unit is responsible for one single logical task, we say it has *high cohesion*.
- *We aim for high cohesion*.
- 'Unit' applies to classes, methods and modules (packages).

High Cohesion

- We aim for high cohesion.
- High cohesion improves *readability* and *reusability*:
  - understand what a class or method does;
  - use descriptive names for variables, methods and classes;
  - reuse classes and methods.

Loose Cohesion

- We aim to avoid loosely cohesive classes and methods.
- Methods perform multiple tasks.
- Classes have no clear identity.

Cohesion at different levels

- Class cohesion:
  - Classes should represent one single, well defined entity.
- Method cohesion:
  - A method should be responsible for one and only one well defined task.

Code Duplication

- Code duplication
  - is an indicator of bad design,
  - makes maintenance harder,
  - can lead to introduction of errors during maintenance.
- Code duplication is *usually a symptom of bad cohesion.*

Code Duplication Example

- See following methods of class Game:
  - printWelcome
  - goRoom
- See section 6.4 on how to fix this.

Adding UP and DOWN

- Add two new directions to the game:
  - "up"
  - "down"
- What do you need to change to do this?
- How easy are the changes to apply thoroughly?

Adding UP and DOWN

- Tow classes must be modified :
  - Room
  - Game
- Room is a short class which stores info about room exits. (section 6.6)
- Game is much longer and complex, and makes heavy use of the exit information about rooms! (section 6.7)

Responsibility-Driven Design

- Question: where should we add a new method (which class)?
- Each class should be responsible for manipulating its own data.
- The class that owns the data should be responsible for processing it.
- *RDD leads to low coupling (+ cohesion).*

Localizing Change

- One aim of reducing coupling and responsibility-driven design is to localize change.
- When a change is needed, as few classes as possible should be affected.

Implicit Coupling

- An even worse form of coupling!
- When the dependence of one class on the internal information of another, is not immediately obvious.
- Example:
  - Add a new command: *look*
  - Need to modify: CommandWords + Game
  - See section 6.9 for details

Thinking Ahead

- When designing a class, we try to think what changes are likely to be made in the future.
- We aim to make those changes easy.

Refactoring

- When classes are maintained, often code is added.
- Classes and methods tend to become longer.
- Every now and then, classes and methods should be *refactored* to maintain cohesion and low coupling.

Refactoring and Testing

- When refactoring code, separate the refactoring from making other changes.
- First do the refactoring only, without changing the functionality.
- Test before and after refactoring to ensure that nothing was broken.
- See section 6.13 for an example.

Enumerated Types

- Uses **`enum`** instead of **`class`** to introduce a type name.
- Their simplest use is to define a set of significant names or constants.
  - Alternative to static **`int`** constants.
  - When the constants' values would be arbitrary.

A basic enum

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

- Each name represents an object of the enum type, e.g., **`CommandWord.HELP`**.
- Enum objects are not created directly.
- Enum definitions can also have fields, constructors and methods.

Design Questions

- Common questions:
  - How long should a class be?
  - How long should a method be?
- These can now be answered in terms of cohesion and coupling.

Design Guidelines

- A method is too long if it does more than one logical task.
- A class is too complex if it represents more than one logical entity.

- Note: these are *guidelines* - they still leave much open to the designer.

Review

- Programs are continuously changed.
- It is important to make this change possible.
- Quality of code requires much more than just performing correct at one time.
- Code must be understandable and maintainable.

Review

- Good quality code avoids duplication, displays high cohesion, and low coupling.
- Coding style (commenting, naming, layout, etc.) is also important.
- There is a big difference in the amount of work required to change poorly structured and well structured code.