Objects First With Java
A Practical Introduction Using BlueJ

Designing classes

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

Main concepts to be covered

- Responsibility-driven design
- Coupling
- Cohesion
- Refactoring

Quality of a class

- What are well designed classes, what are badly designed ones?
- What are the principles we should follow?
- Typically, a poor class design can be done faster – which often makes it more attractive from the short term point of view.
- However, in the long term, things look completely different!

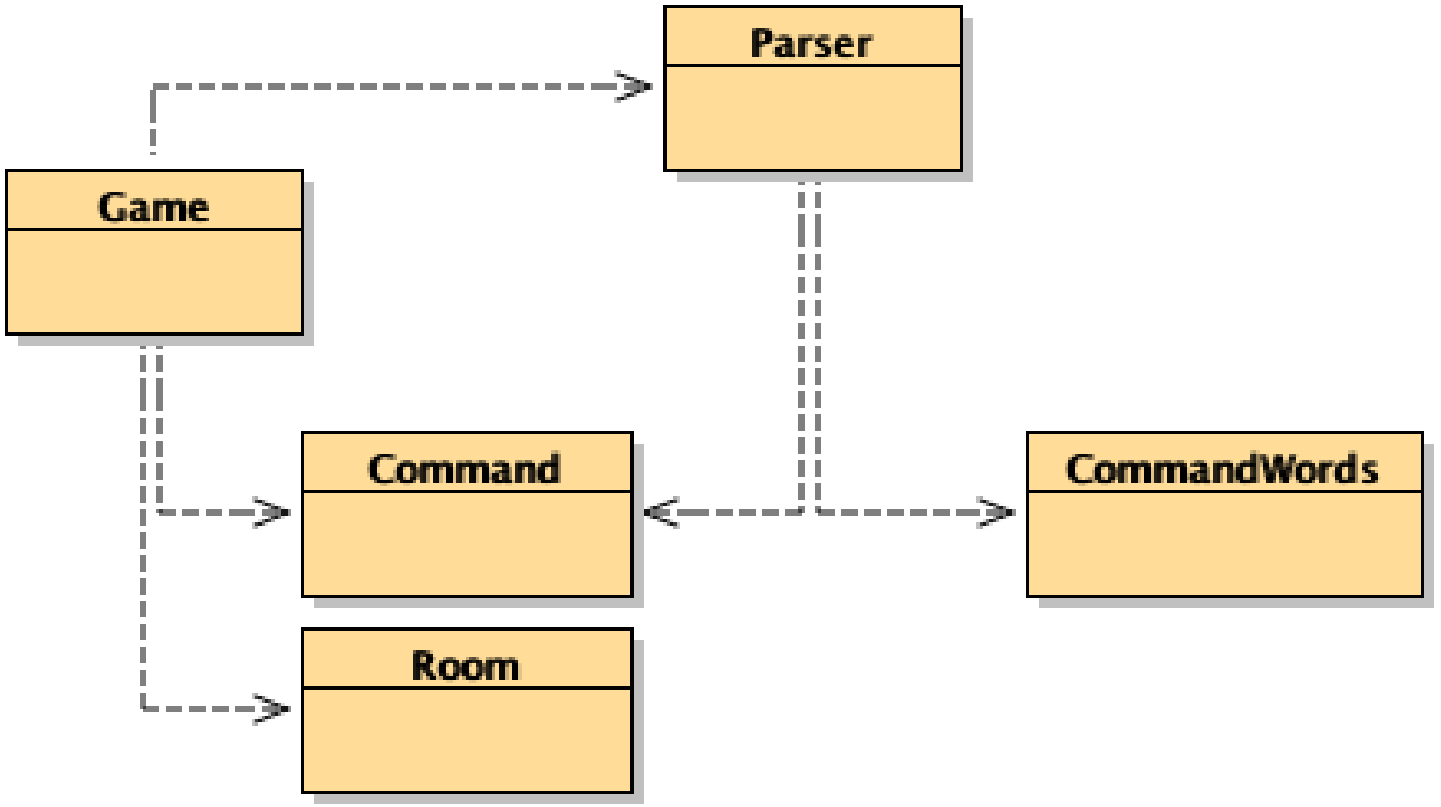
Software changes

- Software is not like a novel that is written once and then remains unchanged.
- Software is extended, corrected, maintained, ported, adapted, re-engineered, …
- The work is done by different people over time (often decades).
- Hence: Functionality is only part of the job – maintainability counts, too!

Change or die

- There are only two options for software:
  - Either it is continuously maintained
  - or it dies.
- Software that cannot be maintained (easily) will be thrown away.

Example (BlueJ): World of Zuul



World of Zuul

- An early adventure game form the early 1970s – finding your way through a complex cave system.
- Very basic structure: 5 classes
  - `CommandWords`: all valid commands (array of strings)
  - `Parser`: reads lines of input and interprets them as commands; creates objects of class `Command`
  - `Command`: a command entered by the user
  - `Room`: a location (rooms have exits leading to other rooms)
  - `Game`: main class, starts the games and then enters a loop to read and execute commands
- Any other ideas for the game design?
- A very nice example for stepwise software design.

Code quality

Two important concepts for quality of code:

- Coupling
- Cohesion

Coupling

- Coupling refers to links or relations between separate units of a program.
  - Remember: Applications are designed as sets of cooperating classes that communicate via well-defined interfaces.
- If two classes depend closely on many details of each other, we say they are *tightly coupled*.
- We aim for *loose coupling*.

Loose coupling

Loose coupling makes it possible to:

- understand one class without reading others;
- change one class without affecting (i.e. having to modify) others – identifying all possible *side effects* may be tedious;
- thus: improve maintainability.

Encapsulation is one strategy to keep coupling loose.

Even worse: implicit coupling

Imagine the following scenario:

- There is a `printHelp` method in `Game` which, among other things, prints a fixed text containing all existing commands.
- Later, you add a new command to your game.
- This is an implicit coupling: the next time a player needs help, the list won't be complete any more.
- Remedy: Don't use a constant text, but let the list of commands be generated by a method in `CommandWords` – this class is responsible of the commands, anyway.
- This will be called responsibility-driven design later!

Encapsulation

- Hide information.
- Fields or methods that do not have to be public should be kept private.

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 task or logical entity, we say it has *high cohesion*.
- Cohesion applies to classes and methods.
- We aim for high cohesion – high cohesion increases the chances of reusability in different contexts.

High cohesion

High cohesion makes it easier to:

- understand what a class or method does;
- identify all relevant pieces in case of a change;
- briefly: to read the code;
- use descriptive names;
- reuse classes or methods.

Cohesion of methods

- A method should be responsible for one and only one well-defined task.
- Cohesive frequently also means short.

Cohesion of classes

- Classes should represent one single, well-defined entity.
- What's bad with non-cohesive classes?
  - Extend World of Zuul: Let each room now contain an item with a description and a weight.
  - Realization: additional fields vs. new class `Item`.
  - What happens if later more than one item can be in one room??

Code duplication

Code duplication

- means to have the same segment of code more than once in one application;
- is an indicator of bad cohesion and design;
- makes maintenance much harder (change all occurrences to avoid inconsistencies);
- can lead to the introduction of errors during maintenance.

Finding code duplicates is one of the challenges of *code re-engineering*.

Code duplication

An example:

- Imagine in class `Game` two methods `LocateYourself` and `GotoRoom`
- Semantics?
- `LocateYourself:`
  - Some tests, and then print the current room.
- `GotoRoom:`
  - Realize the steps, and then print the current room.
- How should these methods *not* be implemented?
- Compare good and bad design, and discuss why!

Responsibility-driven design

- Question: where should we add a new method (where = in 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.
- RDD helps to locate functionality later ("where was this or that defined?").

Localizing change

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

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.
- Example:
  - In our game, so far all output is done via `System.out.println` statements.
  - What should be changed, if we thought about placing response in a text field in a window??

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

  (automated tests).

Design questions

Common questions:

- How long should a class be?
- How long should a method be?

- Can not be answered in terms of lines of code.
- 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, 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.