Software Development 2

Object Oriented Programming and Designing Classes

F27SB

Aim of this lecture

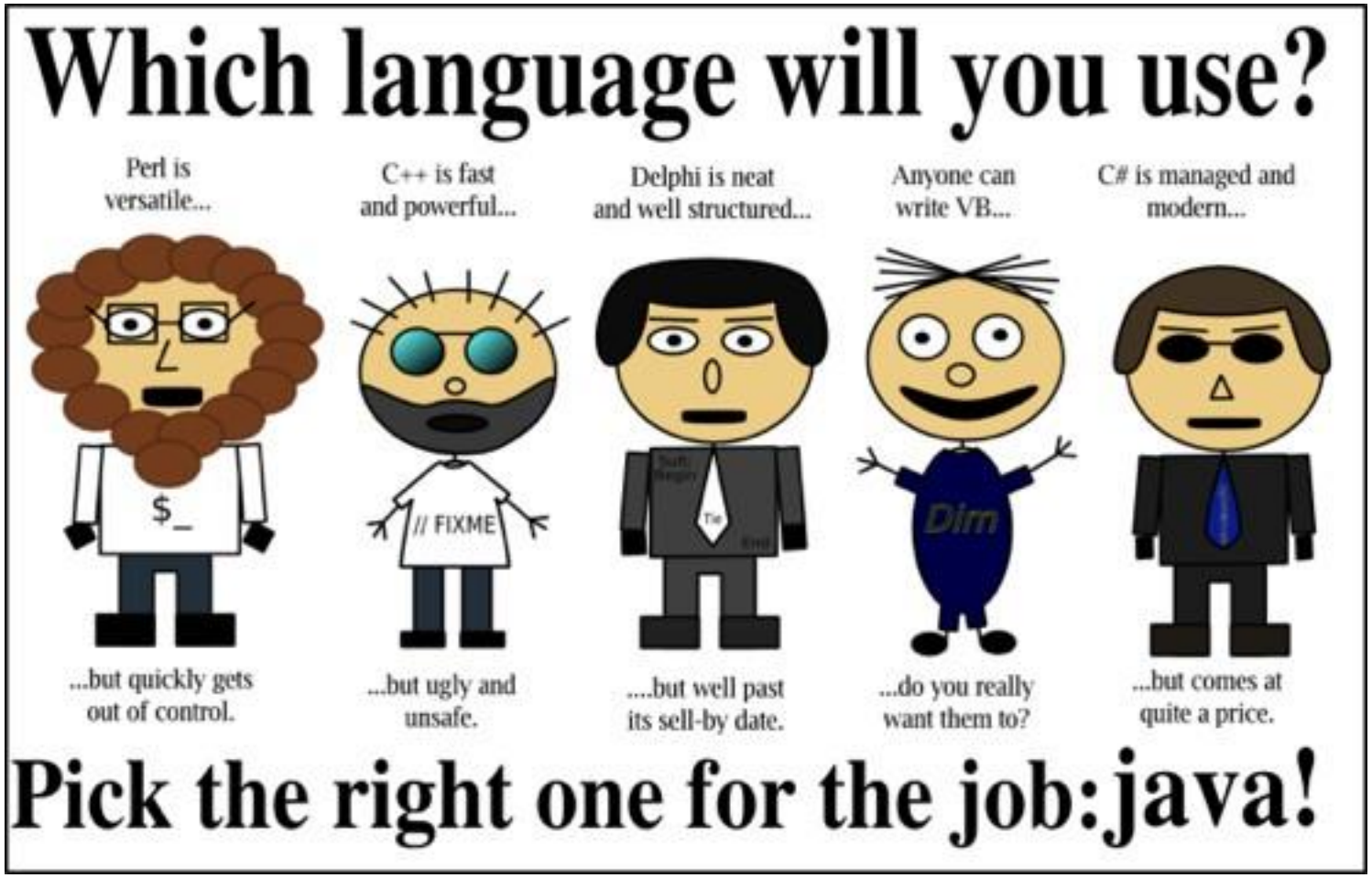
- Understand how to write classes in a way that they are:
  - easy to understand
  - maintainable
  - reusable.

→ OOP principles

Isn’t python much better?

WHY JAVA?

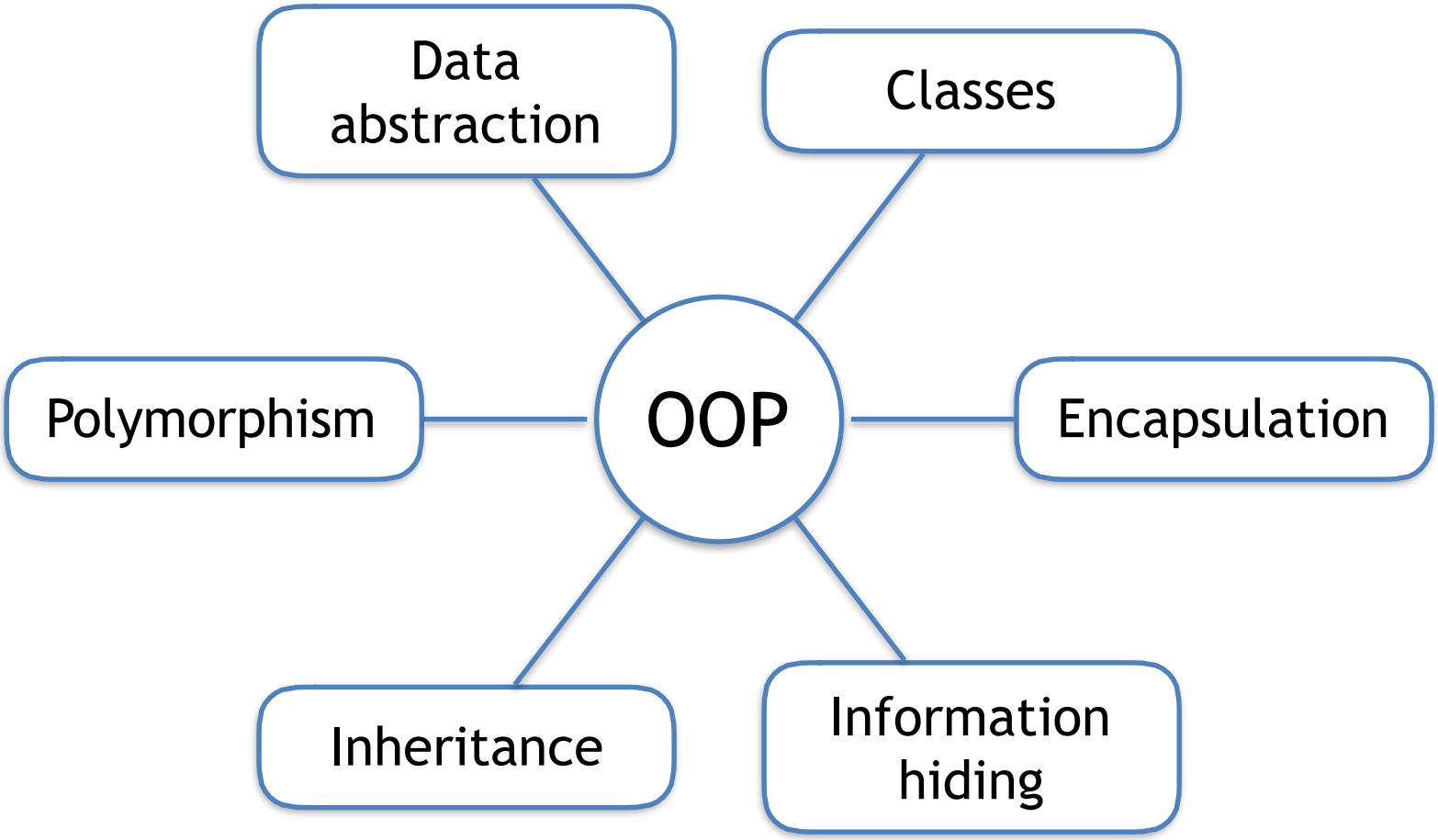
A matter of faith? No “one fit for all”



Is Java “the best”?

- The “best” programming language depends on the problem you are trying to solve.
- Java is flexible and widely used.
- However, it’s slower than C++, for example.
- It also requires more programming “overload” (e.g. type checks) than simple scripting languages like Python or Pearl.
- We choose Java for you to learn an ***Object Oriented Programming (OOP)*** Language.

OOP fundamental concepts



To be discussed throughout this course!

Change or die

Why is all this important?

- There are only two options for software:
  - Either it is continuously maintained
  - Or it dies

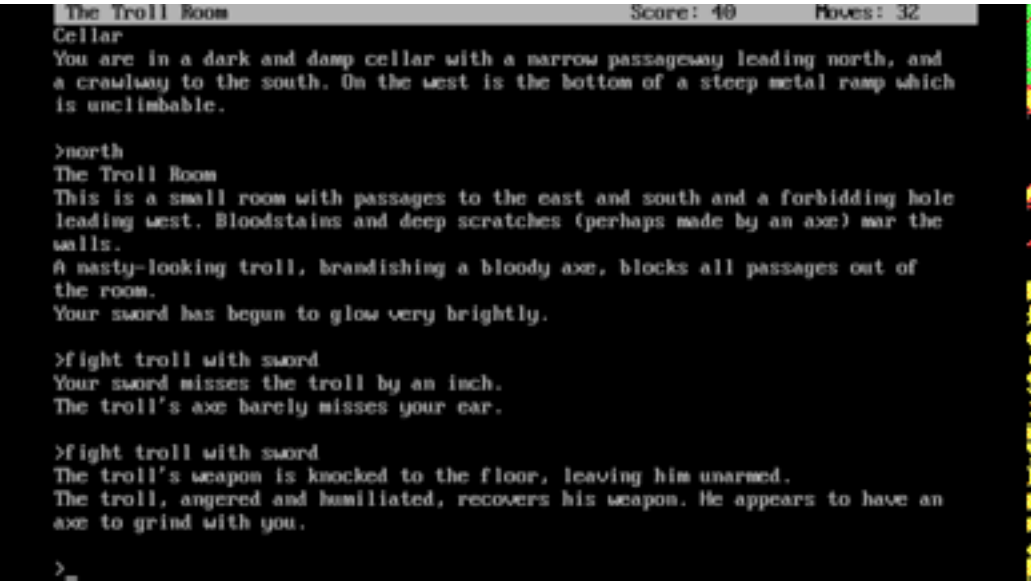
→ Software that cannot be maintained will be thrown away.

Fight troll with sword

TEXT ADVENTURES

Example: The World-of-Zuul

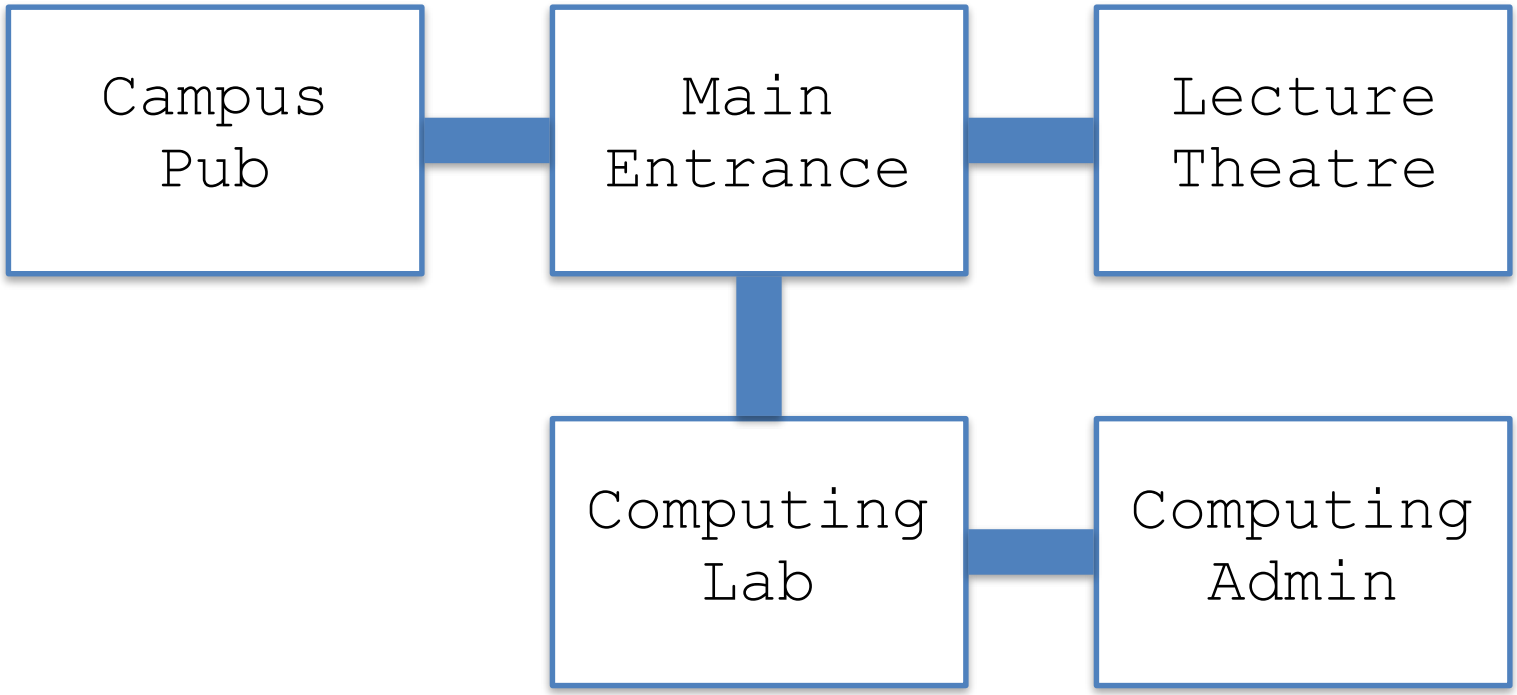
- Inspired by the first (text-based!) adventure games ADVENT (1976/77).
- Navigate using text command (“go east”).
- Also see: “Interactive fiction”.



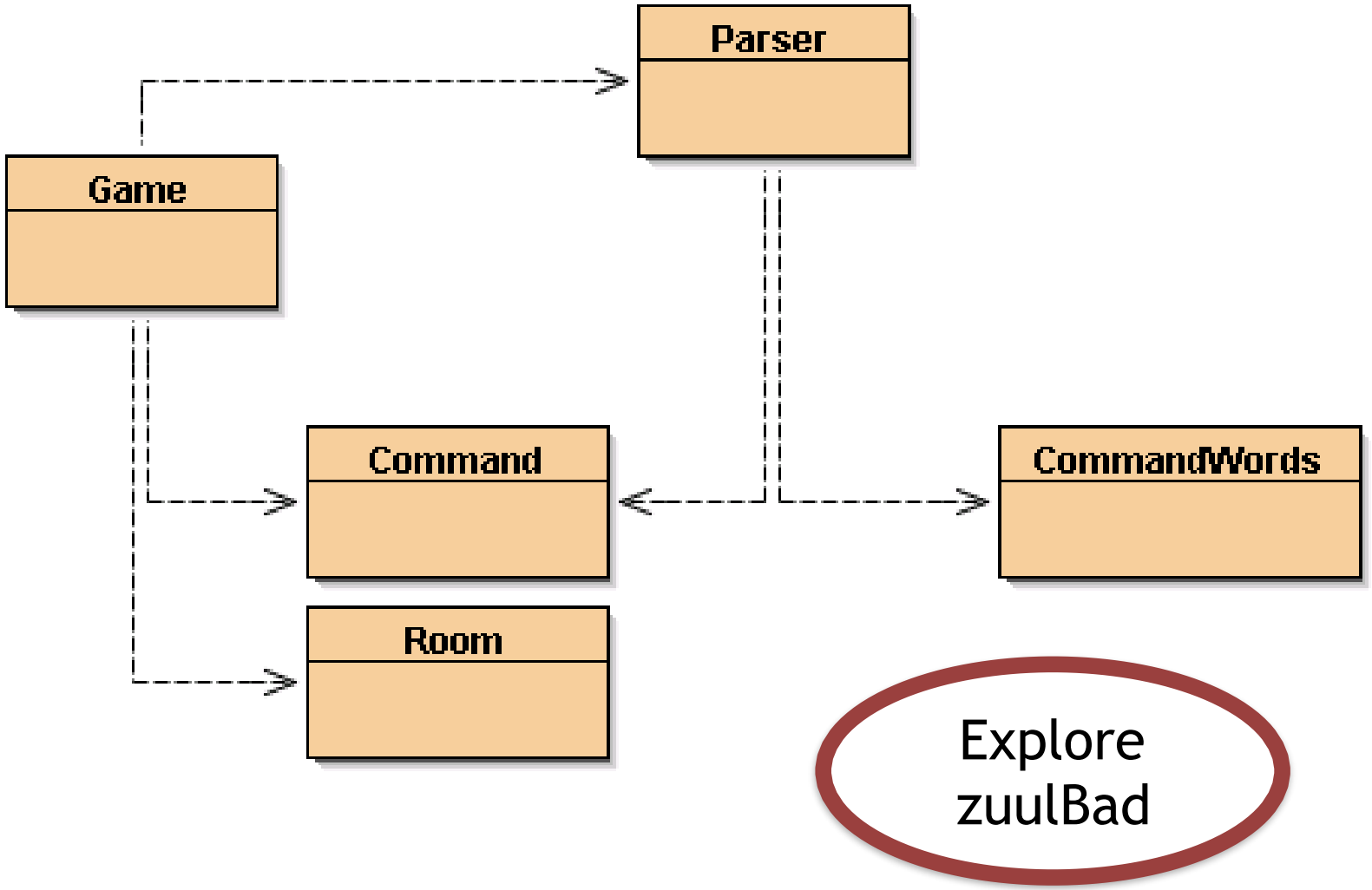
Text-based Adventure Games: Colossal Cave Adventure (1977/79)

```
.RUN ADV11

WELCOME TO ADVENTURE!!  WOULD YOU LIKE INSTRUCTIONS?


YES
SOMEWHERE NEARBY IS COLOSSAL CAVE, WHERE OTHERS HAVE FOUND
FORTUNES IN TREASURE AND GOLD, THOUGH IT IS RUMORED
THAT SOME WHO ENTER ARE NEVER SEEN AGAIN. MAGIC IS SAID
TO WORK IN THE CAVE.  I WILL BE YOUR EYES AND HANDS. DIRECT
ME WITH COMMANDS OF 1 OR 2 WORDS.
(ERRORS, SUGGESTIONS, COMPLAINTS TO CROWTHER)
(IF STUCK TYPE HELP FOR SOME HINTS)


YOU ARE STANDING AT THE END OF A ROAD BEFORE A SMALL BRICK
BUILDING . AROUND YOU IS A FOREST. A SMALL
STREAM FLOWS OUT OF THE BUILDING AND DOWN A GULLY.


GO IN
YOU ARE INSIDE A BUILDING, A WELL HOUSE FOR A LARGE SPRING.


THERE ARE SOME KEYS ON THE GROUND HERE.


THERE IS A SHINY BRASS LAMP NEARBY.


THERE IS FOOD HERE.


THERE IS A BOTTLE OF WATER HERE.
```

The Current World of Zuul



World of Zuul



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:** Recognised user commands.

Zuul-bad vs. Zuul-better

- Same functionality.
- Different class structure.
- Why is this important?
- Has a big impact on how easy it is to maintain and extend the game!
- How can we measure it?

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

- 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.
- 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.

Encapsulation

- = hiding information from view.
- Only information about WHAT a unit, e.g. class, can do should be visible from the outside.
- The HOW should be hidden/ “encapsulated”.
- Reduces coupling.

Remember **delegation** from last week?

```
public int getNumberOfFiles() {
        return files.size();
}
```

COHESION

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 makes it easier to:
  - understand what a class or method does;
  - use descriptive names for variables, methods and classes;
  - reuse classes and methods.
  - Read other people's code.

Loose cohesion

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

Cohesion applied at different levels

- Class level:
  - Classes should represent one single, well defined entity.

- Method level:
  - A method should be responsible for one and only one well defined task.

Example for Method Cohesion

```
 public void play() {
        printWelcome();
   …
}
```

```
private void printWelcome(){
      System.out.println(
       "Welcome to the World of Zuul!");
         ...
}
```

So that’s what the text adventure was for…

DESIGN EXAMPLES AND GUIDELINES

Design guidelines

- A method is too long if it does more then 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.

Responsibility-driven design (RDD)

- 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.

Good Example

```
private static final String[]
validCommands =
{
        "go", "quit", "help“
};
```

One central place to store all valid command words.

Localising change

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

Code duplication

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

Extending the World-of-Zuul

- Add two new directions to the 'World of Zuul' for multi-storey buildings, e.g. dungeons:
  - "up"
  - "down"
- What do you need to change to do this?
- How easy are the changes to apply thoroughly?

Needs changing:

- Room:
  - setExits()
- Game:
  - printWelcome()
  - goRoom()
  - createRoom()

- High Coupling between Room and Game
- Loose cohesion of Room
- Code duplication, e.g. printWelcome() and goRoom()

Example: Room Exists in Zuul-Bad

- Single variables getting used in various methods and classes

→ hard to change or extend functionality.

Ideas fro improvement

FROM TIGHT COUPLING TO LOOSE COUPLING

What else is badly designed?

```
public String description;
public Room northExit;
public Room southExit;
public Room eastExit;
public Room westExit;
```

Fields in Room are public!

In Room.java:

```
private HashMap<String, Room> exits;

public void setExit(String direction,
Room neighbor)
{
      exits.put(direction, neighbor);
}
```

In Game.java:

```
outside.setExit("east", theater);
outside.setExit("south", lab);
outside.setExit("west", pub);
```

Summary

- 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.
- Next: Safe Refactoring.

Past Exam Questions

Consider the following method:

```
public void createMembership(String name, String id)
{           name.trim();
            if (name.length() > 0){
                        this.name= name;
            }
            if(ID.length() < ID_MAX_LENGTH) {
                        this.id =id;
            }
            Member m = new Member(this.id, this.name);
            List<Member> existingMembers  = this.database.getExistingMembers();

            for (int i = 0; i < existingMembers.size(); i++)
            {
                if(this.id == existingMembers.get(i).getID() )
                        {System.out.println("This member already exists “+ “in the database!");
                        }
                        else
                        {
                                    existingMember.add(m);
                        }
            }
}
```

What is cohesion? Discuss the above method with respect to this concept.

Homework

- Read Chapter 6 “Designing classes”.
- Have a look at SD3 lecture 1 for Unit tests again