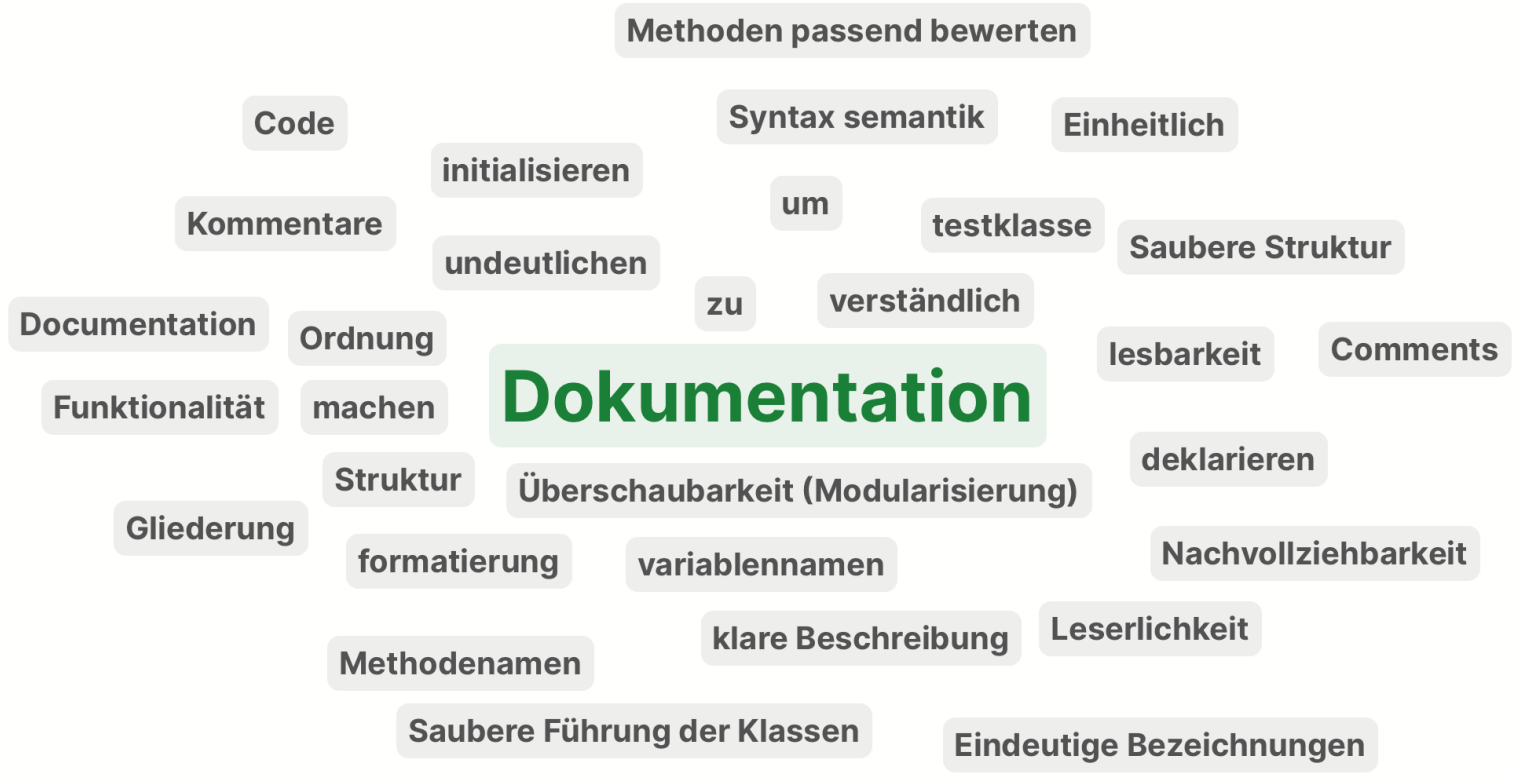
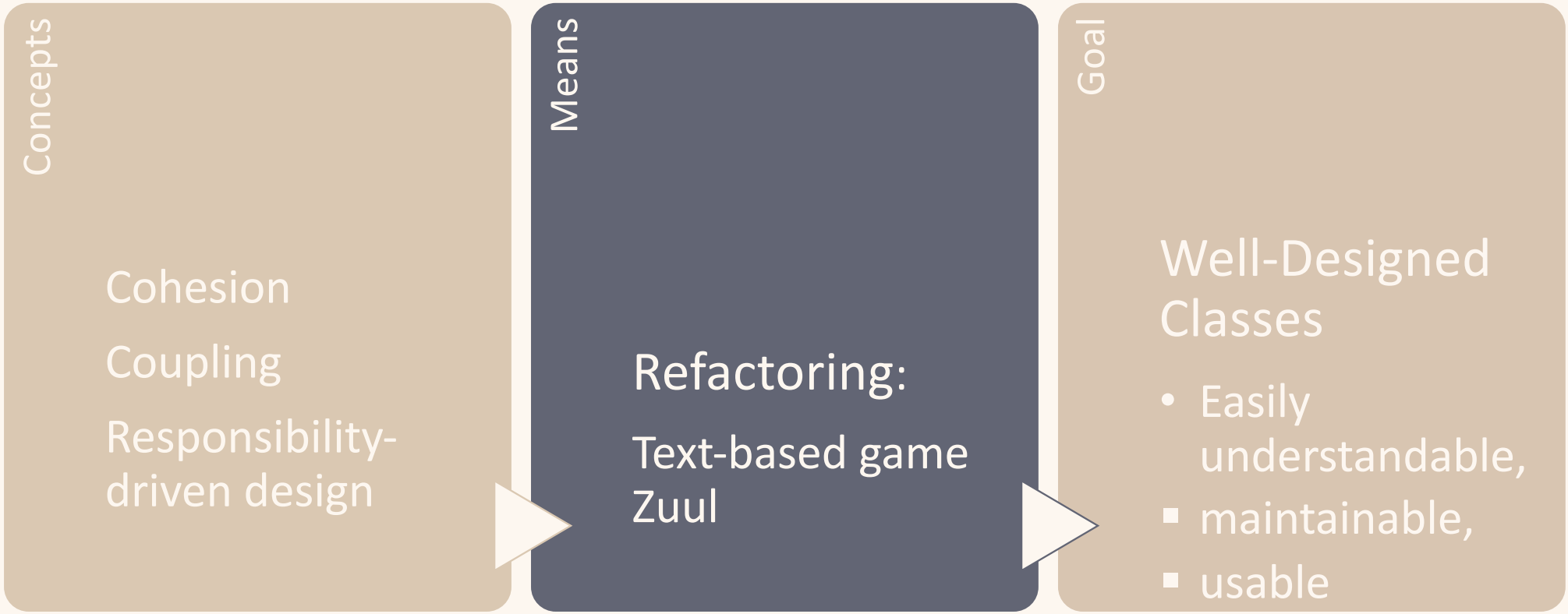
Active poll 11

Wonach bewerten Sie die Qualität von Source Code?



Goal, concepts, means



Motivation: 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).

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

Factors of software change

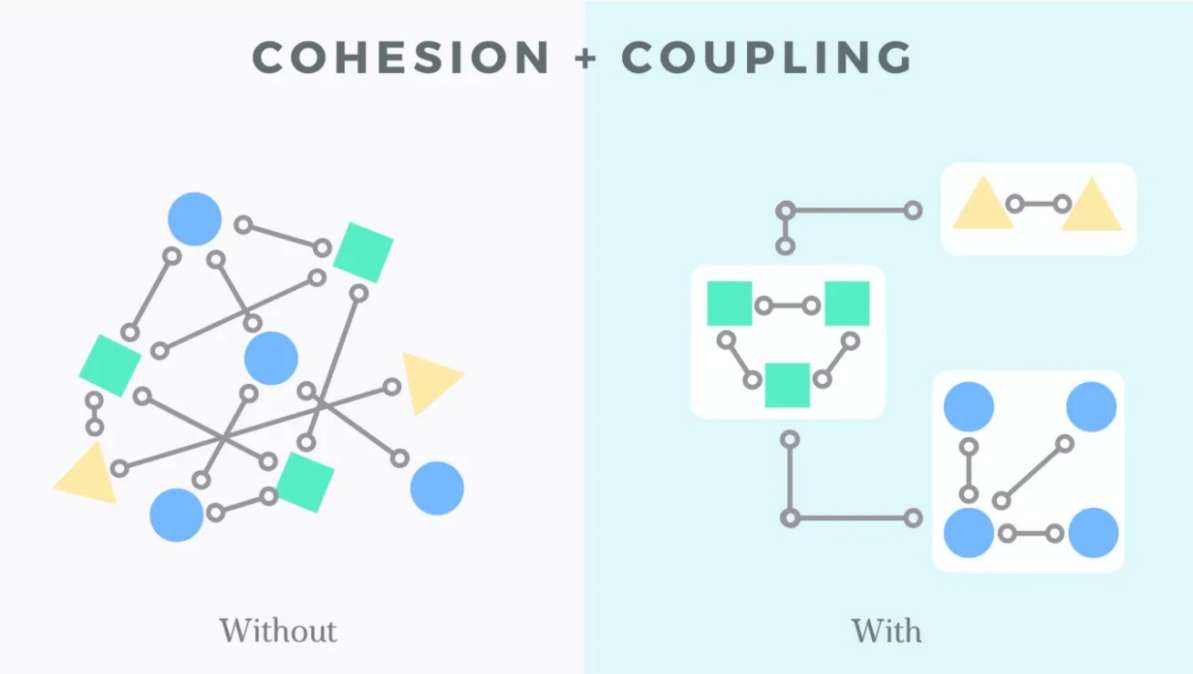
Table 3. Factors of software changes.

RQ	Effect	Remedies
RQ2 Gathering and changing user requirements	16: Strongly affects	14: multiple meetings with users 4: use mock-ups 4: use stories 4: use control panel 3: train devs in domain 3: study the user process most: changes in req are inevitable, just give more time.
RQ7 Add new feature	16: Strongly affects	4: improve requirements gathering
RQ1 Apply standard methodologies for SDLC	14: Strongly affects 1: Affects 1: Somewhat affects	8: apply SDLC to the letter 2: dedicate bug fixing iterations
RQ4 Bugs	13: Strongly affects 3: Affects	fairly common suggestions (see text)
RQ6 Refactoring	10: Strongly affects 3: Affects 3: Somewhat affects	5: involve reviewer and QA in dev team 5: hire experienced devs and train existing devs 5: build independent software services 2: use good design structure
RQ9 Lack of HCI skills	6: Strongly affects 6: Affects 4: Somewhat affects	9: employ UX designers 3: use mock-ups 3: run beta to get user feedback 3: reduce user clicks 2: reduce input fields, links, etc. 2: use validation messages
RQ3 Developer Experience	4: Strongly affects 12: Affects	8: involve reviewer and QA in dev team 4: use comments 4: expand business experience of devs. most: teams should mingle and loads should be balanced.
RQ8 Reduce resources	1: Affects 7: Somewhat affects 6: Neutral 2: Does not affect	
RQ5 Upgrade environment and development framework	5: Somewhat affects 7: Neutral 4: Does not affect	

Adorjan, Alejandro & Solari, Martín & Vallespir, Diego. (2020
Practices: Perspective of Team Leaders. 213-221.
10.1109/CLEI52000.2020.00032.

Main Concepts to Be Covered

- Coupling (aiming for loose coupling)
- Cohesion (aim for high cohesion)
- Responsibility-driven design
- By means of : Refactoraing



https://blog.stackademic.com/low-coupling-and-high-cohesion-in-system-development-03ecdddf5ba6

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

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

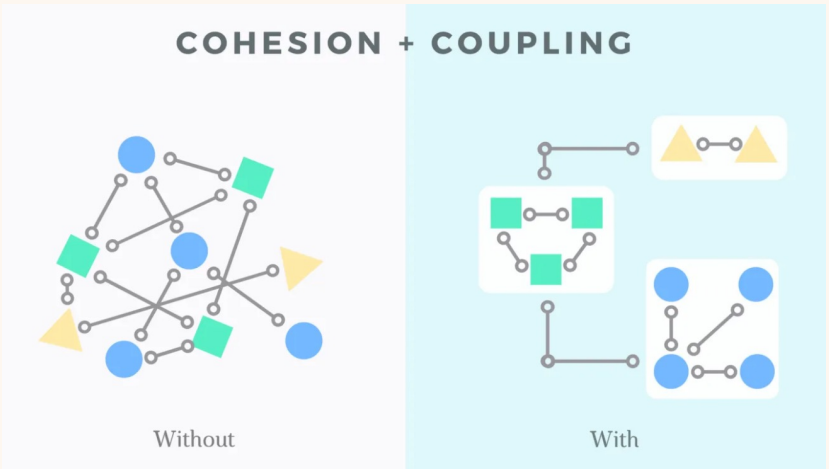
“In der objektorientierten Programmierung beschreibt Kohäsion, wie gut eine Programmeinheit eine logische Aufgabe oder Einheit abbildet. In einem System mit starker Kohäsion nimmt jede Programmeinheit (eine Methode, eine Klasse oder ein Modul) genau eine wohldefinierte Aufgabe oder Einheit wahr.”

https://de.wikipedia.org/wiki/Koh%C3%A4sion_(Informatik)

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Coupling

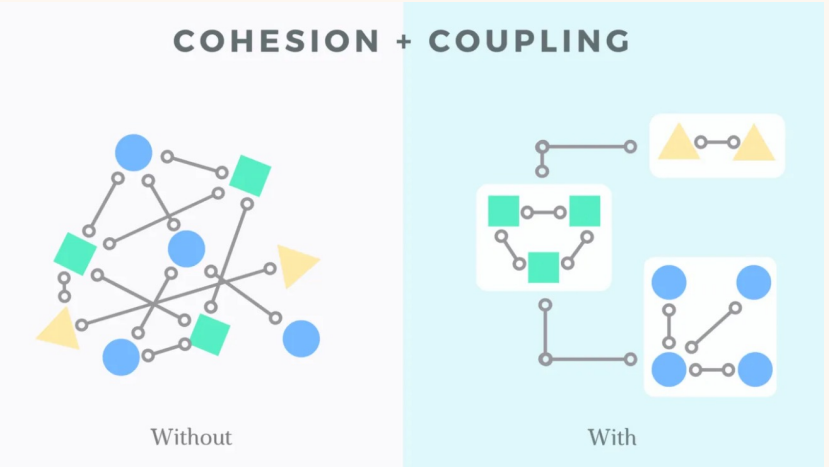
- Coupling refers to links between separate units of a program.
- If two classes depend closely on many details of each other, we say they are *tightly coupled*. -> difficult to change
- *We aim for loose coupling. -> easy to change*
- A class diagram provides hints at where coupling exists.



© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

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).



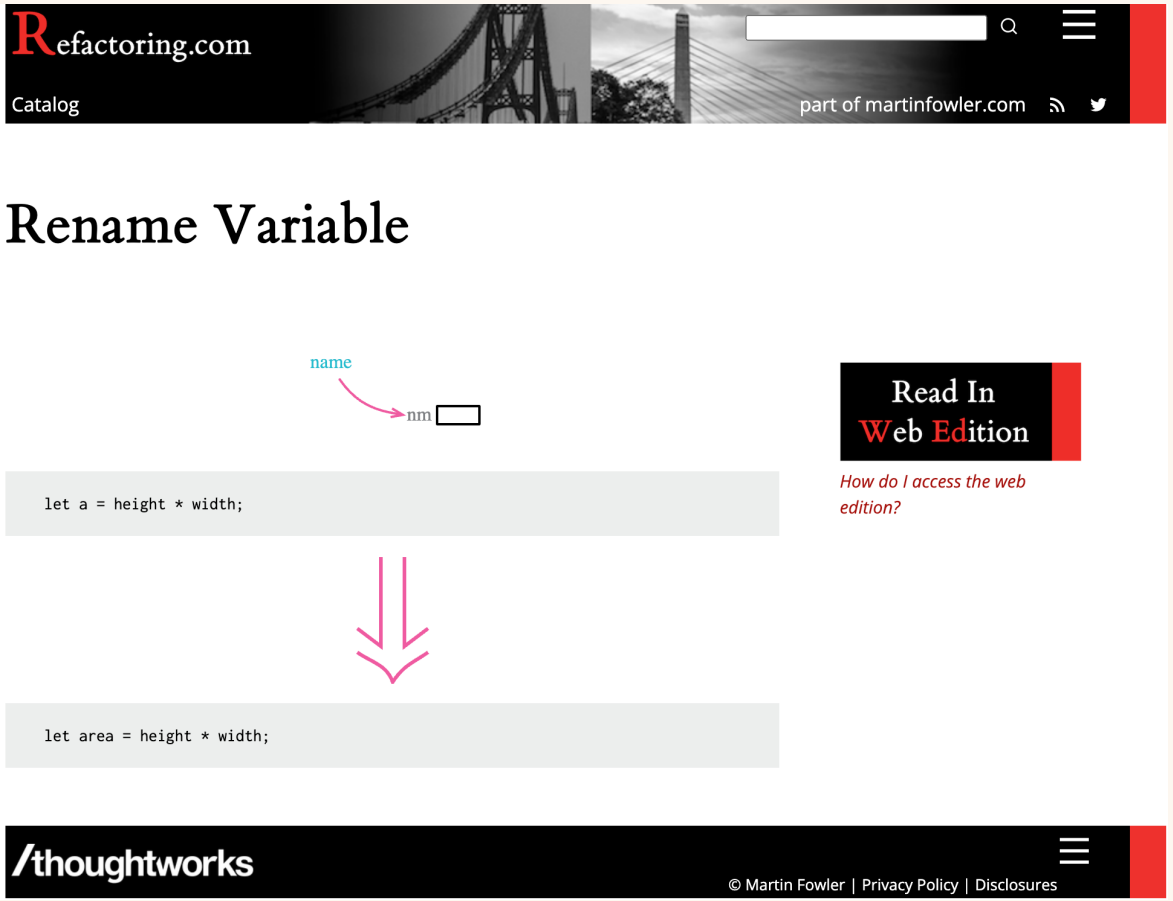
Refactoring

"Refactoring is a controlled technique for improving the design of an existing code base.

Its essence is applying a series of small behavior-preserving transformations, each of which "too small to be worth doing".

However the cumulative effect of each of these transformations is quite significant. By doing them in small steps you reduce the risk of introducing errors."

(Martin Fowler)



https://martinfowler.com/books/refactoring.html

https://refactoring.com/catalog/

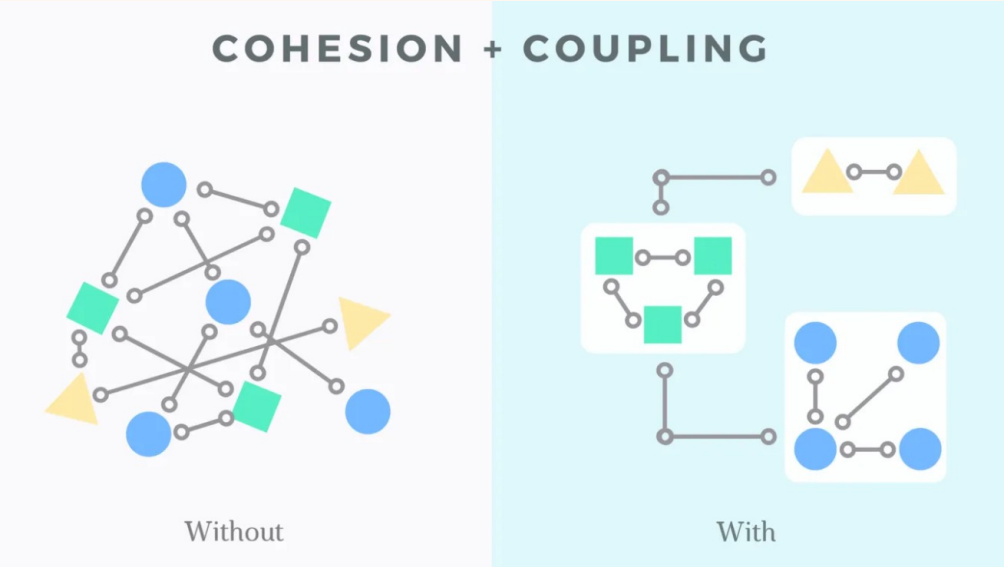
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Refactoring

When classes are maintained code is usually added.

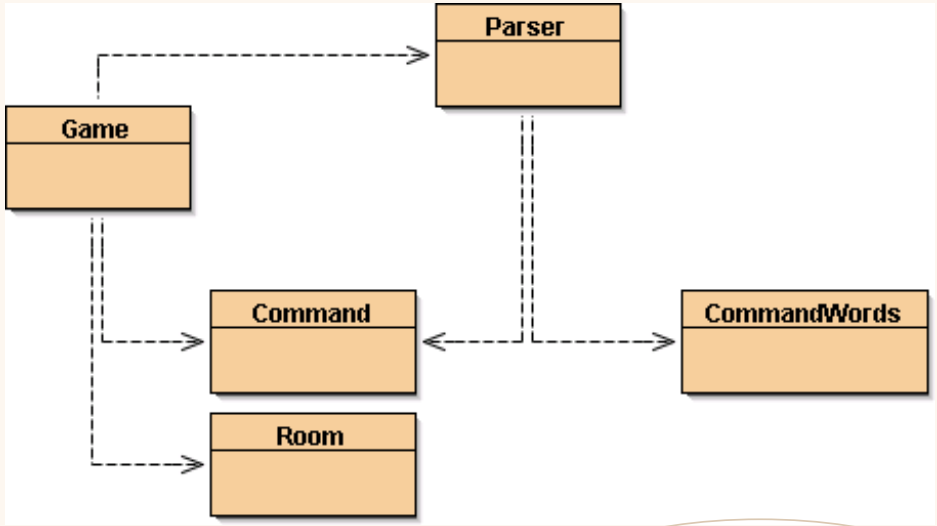
Classes and methods tend to become longer.

Every now and then, classes and methods should be *refactored* to maintain cohesion and low coupling.



© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

World of Zuul



Explore `zuul-bad`

Explore Zuul project

a) What commands does the game accept?

b) What does each command do?

c) How many rooms are in the scenario?

e) Draw a map of the existing rooms.

Explore Zuul project

1. Look at all the classes and mark the method
signatures such that you can find them more easily later on.
2. Write down the responsibilities of all the Classes in your own words:
a. Game, b. Room, c. Parser, d. Command, e. CommandWords

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

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.

Looking into Zuul project bad

How does bad design look?

Bad design involves something deeper than simply the way a class looks or how well it's documented….

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Code duplication

An indicator of bad design.

Makes maintenance harder.

Can lead to the introduction of inconsistencies and errors during maintenance/modification.

Occurs at both method and class level.

➔ Compare printWelcome() and goRoom()

➔ printLocationInfo(), to be used by the other methods

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

A worked example to test quality

Add two new directions to the 'World of Zuul':

- “up”
- “down”

What do you need to change to do this?

How easy are the changes to apply thoroughly?

Explore Zuul project

3. Have you found any code duplications? How could you remove them?

4. As with all successful Software, we will want to change and extend the game. (Erweiterbarkeit) We will add new directions (e.g. up and down) and new commands.

- a. Mark and list all places in the source code that need to be adapted to add a new direction.
- b. Which class should be responsible for “knowing” about directions?

Refactoring poor Code

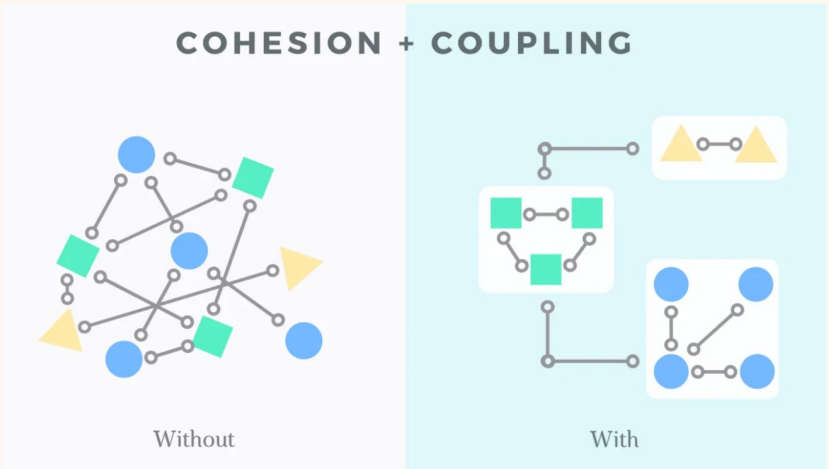
- Code duplication -> tight coupling ☹

- Use of Room fields in Game class -> tight coupling ☹
  - Public fields
  - Solution: private fields and getter methods -> Encapsulation

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

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*. -> difficult to change
- *We aim for loose coupling. -> easy to change*
- A class diagram provides hints at where coupling exists.



© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

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.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Reducing coupling

Encapsulation supports loose coupling.

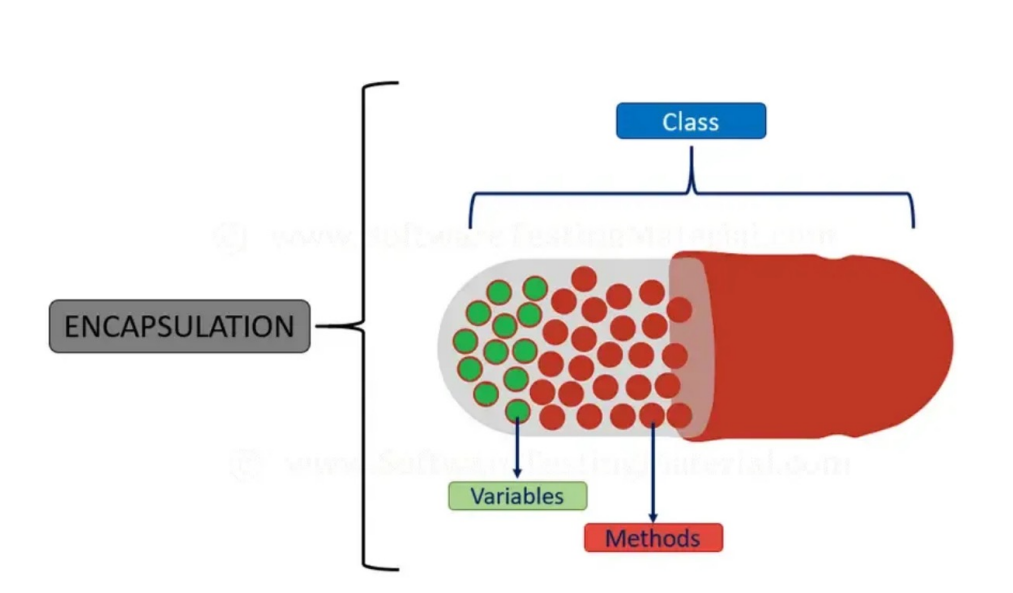
- private elements cannot be referenced from outside the class.
- Reduces the impact of internal changes.

Responsibility-driven design supports loose coupling.

- Driven by data location.
- Enhanced by encapsulation.

"the bundling of data and methods

that operate on that data within a single unit"



https://medium.com/@udaypatil318/encapsulation-in-java-da8d1c3c9caa

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Public vs private access modifier (Zugriffsmodifikator)

Public elements are accessible to objects of other classes:

- Fields, constructors and methods
- Only methods **that are intended for other classes** should be public.

Fields should not be public.

Private elements are accessible only to objects of the same class.

- to break up a large task into several smaller ones
- Subtask needed for several methods of a class

Protected will be discussed later

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Information hiding

Data belonging to one object is hidden from other objects.

Know *what* an object can do, not *how* it does it.

Information hiding increases the level of *independence*.

Independence of modules is important for large systems and maintenance.

Refactor: encapsulation

In Room class

- Change fields to private
- Add getterMethod

encapsulation

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

    Existing methods unchanged.

    public Room getExit(String direction)
    {
        if(direction.equals("north")) {
            return northExit;
        }
        if(direction.equals("east")) {
            return eastExit;
        }
        if(direction.equals("south")) {
            return southExit;
        }
        if(direction.equals("west")) {
            return westExit;
        }
        return null;
    }
}
```

Refactor: encapsulation

I Game class

- Access with getter Method

```
private void goRoom(Command command)
{
    if(!command.hasSecondWord()) {
        // if there is no second word, we don't know where to g
        System.out.println("Go where?");
        return;
    }

    String direction = command.getSecondWord();

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

    /*if(direction.equals("north")) {
        nextRoom = currentRoom.getExit("north");
    }
    if(direction.equals("east")) {
        nextRoom = currentRoom.getExit("east");
    }
    if(direction.equals("south")) {
        nextRoom = currentRoom.getExit("south");
    }
    if(direction.equals("west")) {
        nextRoom = currentRoom.getExit("west");
    }
    */
    if (nextRoom == null) {
        System.out.println("There is no door!");
    }
    else {
        currentRoom = nextRoom;
        printLocationInfo();
    }
}
```

Refactor:

Exercise 8.7 Make a similar change to the **printLocationInfo** method of **Game** so that details of the exits are now prepared by the **Room** rather than the **Game**. Define a method in **Room** with the following header:

```
/**
 * Return a description of the room's exits,
 * for example, "Exits: north west".
 * @return A description of the available exits.
 */
public String getExitString()
```

```java
/**
 * Return a description of the room's exits
 * @return A String with room exits
 */

public String getExitString(){

    String returnString = "Exits: ";

    if(getExit("north") != null) {
        returnString +="north";
    }

    if(getExit("east") != null) {
       returnString +="east";
    }
    if(getExit("west") != null) {
        returnString +="west";
    }
    if(getExit("south") != null) {
        returnString +="south";
    }

    return returnString;

}
```

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

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.

Implementation of Loose Coupling in Zuul: Replaced public field access with method calls to decouple internal representation from interface.

Benefit of Loose Coupling: Allows understanding and changing one class without affecting others.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Implicit coupling

Linkage not necessarily expressed via data and method interactions.

Can arise from assumptions about the way a class is implementing.

Satisfactory resolution might require a small increase in explicit coupling.

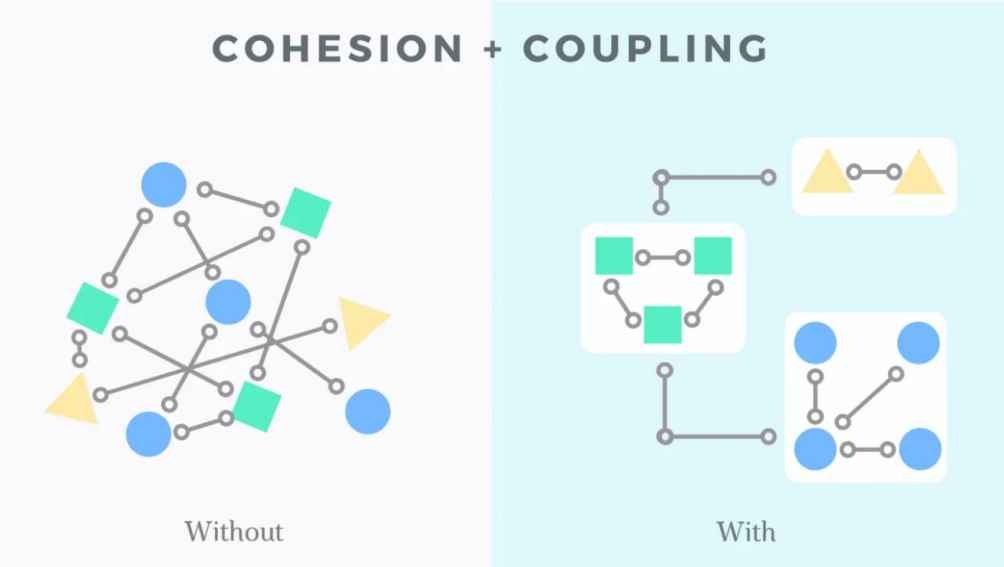
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Refactoring

When classes are maintained code is usually added.

Classes and methods tend to become longer.

Every now and then, classes and methods should be *refactored* to maintain cohesion and low coupling.



Refactor: replace ducplicatio

Use a Hashmap to store the exits of a Room!

Change room fields -> Hashmap

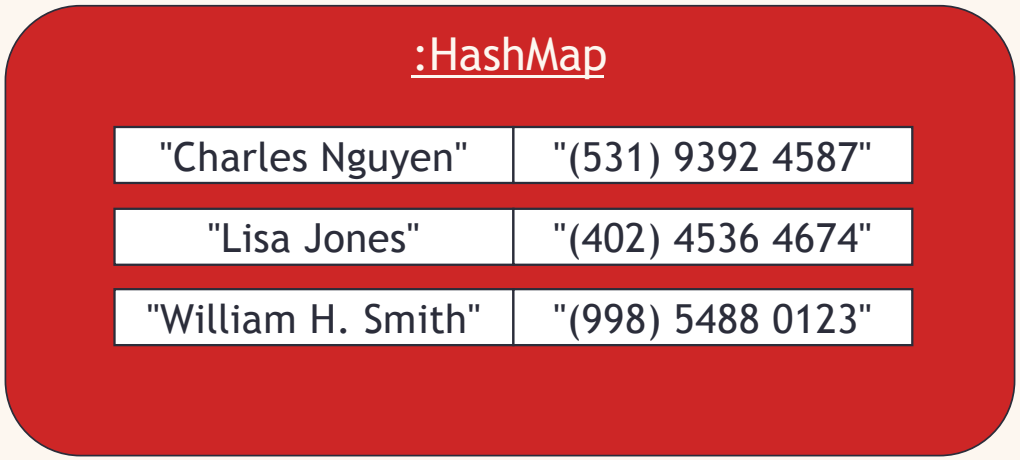
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Reminder

Using maps

A map with strings as keys and values

One way lookup!



© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

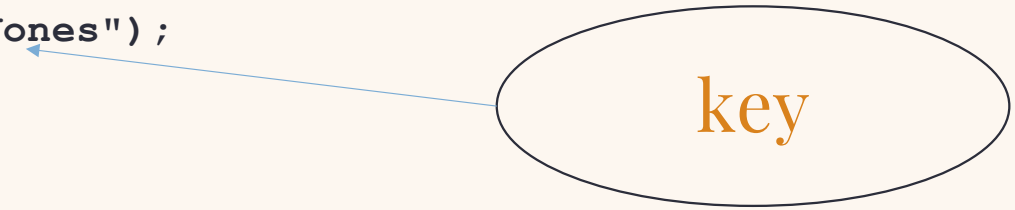
Reminder

Using maps

```
HashMap <String, String> contacts = new HashMap<>();

contacts.put("Charles Nguyen", "(531) 9392 4587");
contacts.put("Lisa Jones", "(402) 4536 4674");
contacts.put("William H. Smith", "(998) 5488 0123");

String number = contacts.get("Lisa Jones");
System.out.println(number);
```



Reminder

Practice

a) What happens if you add an entry to a map with a key that already exists in the map?
   - It overwrites the previous value associated with the key.

b) What happens when you add an entry to a map with two different keys?
   - Both values stay in the map. HashMaps only uses the key to distinguish entries - not the values.

c) How do you check whether a given key is contained in a map? (Give a java code example)
   - myMap.containsKey(“key”);

d) What happens when you try to look up a value and the key does not exist in the map?
   - Returns null

e) How do you check how many entries are contained in a map?
   - myMap.size()

f) How do you print out all keys currently stored in a map?
   - ```
     for(String name : myMap.keySet()) {
         System.out.println(name);
     }
     ```
     ```

Refactor: exits fields stored in HashMap

- Declare
- Initialize
- Fill

```
import java.util.HashMap;

public class Room
{
    private String description;
    private HashMap<String, Room> exits;

    /*private Room northExit;
    private Room southExit;
    private Room eastExit;
    private Room westExit;*/

    /**
     * Create a room described "description". Initi
     * no exits. "description" is something like "a
     * "an open court yard".
     * @param description The room's description.
     */
    public Room(String description)
    {
        this.description = description;
        exits= new HashMap<>();
    }
```

```
    /**
     * Define the exits of this room.  Every direction either leads
     * to another room or is null (no exit there).
     * @param north The north exit.
     * @param east The east east.
     * @param south The south exit.
     * @param west The west exit.
     */
    public void setExit(String direction, Room exit)
    {
        exits.put(direction,exit);
        /*
        if(north != null) {
            northExit = north;
        }
        if(east != null) {
            eastExit = east;
        }
        if(south != null) {
            southExit = south;
        }
        if(west != null) {
            westExit = west;
        }*/
    }
```

Refactor: exits fields stored in HashMap

- Get

```
public Room getExit(String direction){

    return exits.get(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;
    }
    return null;
    */
}
```

```
public String getExitString(){

    String returnString = "Exits: ";
    for(String exit : exits.keySet()) {
        returnString += " "+ exit;
    }

    /*

    if(getExit("north") != null) {
        returnString +="north";
    }

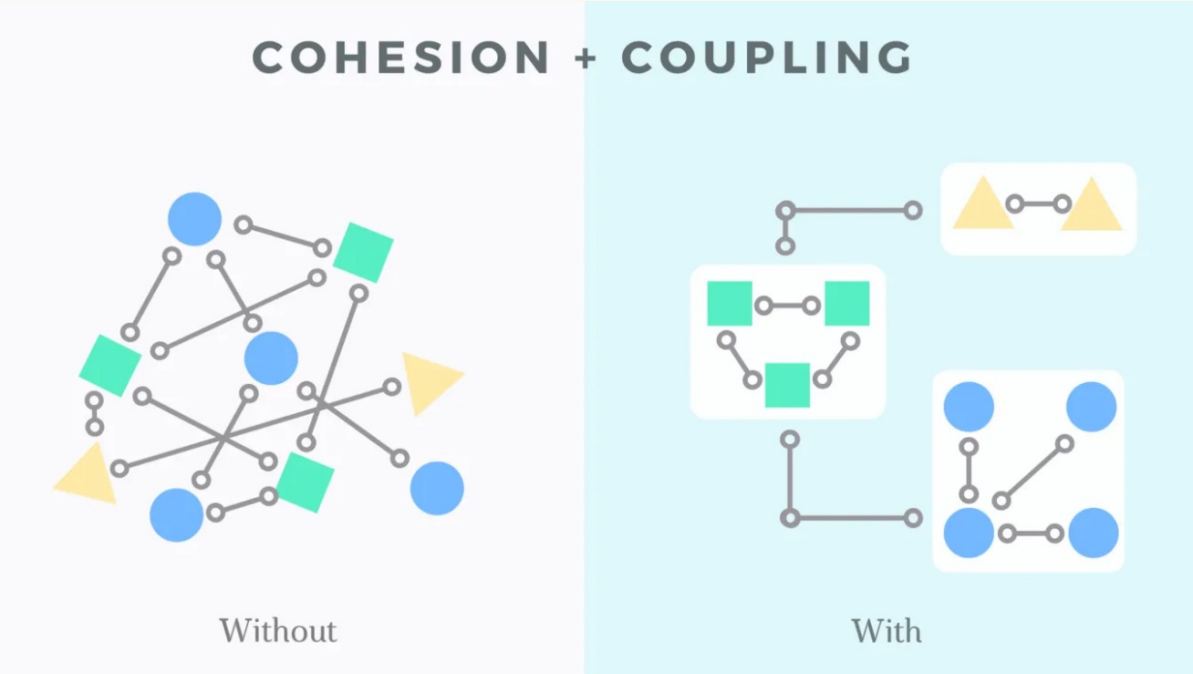
    if(getExit("east") != null) {
       returnString +="east";
    }
    if(getExit("west") != null) {
        returnString +="west";
    }
    if(getExit("south") != null) {
        returnString +="south";
    }

    */
    return returnString;

}
```

Main Concepts to Be Covered

- Coupling (aiming for loose coupling)
- Cohesion (aim for high cohesion)
- Responsibility-driven design
- Refactoring



https://blog.stackademic.com/low-coupling-and-high-cohesion-in-system-development-03ecdddf5ba6

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Code duplication

An indicator of bad design.

Makes maintenance harder.

Can lead to the introduction of inconsistencies and errors during maintenance/modification.

Occurs at both method and class level.

```
public class Room
{
    private String description;
    private HashMap<String, Room> exits;

    /*private Room northExit;
    private Room southExit;
    private Room eastExit;
    private Room westExit;*/
```

```
public void setExit(String direction, Room exit)
{
    exits.put(direction,exit);
    /*
    if(north != null) {
        northExit = north;
    }
    if(east != null) {
        eastExit = east;
    }
    if(south != null) {
        southExit = south;
    }
    if(west != null) {
        westExit = west;
    }*/
}
```

```
public String getExitString(){

    String returnString = "Exits: ";

    for(String exit : exits.keySet()){
            returnString += " "+exit;
    }
    /*
     if(getExit("north") != null) {
        returnString += "north ";
    }
    if(getExit("east") != null) {
        returnString += "east ";
    }
    if(getExit("south") != null) {
        returnString += "south ";
    }
    if(getExit("west") != null) {
        returnString += "west ";
    }*/

    return returnString;
}
```

High cohesion

One method, one task only

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

    // Enter the main command loop.  Here we repeatedly read commands and
    // execute them until the game is over.

    boolean finished = false;
    while (! finished) {
        Command command = parser.getCommand();
        finished = processCommand(command);
    }
    System.out.println("Thank you for playing.  Good bye.");
}

/**
 * Print out the opening message for the player.
 */
private void printWelcome()
{
    System.out.println();
    System.out.println("Welcome to the World of Zuul!");
    System.out.println("World of Zuul is a new, incredibly boring adventure game.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    this.printLocationInfo();
}

private void printLocationInfo(){
    System.out.println("You are " + currentRoom.getDescription());
    currentRoom.getExitString();
```

Explicit Coupling

Compiler supports detecting change

```
private void printLocationInfo(){
    System.out.println("You are " + currentRoom.getDescription());
    currentRoom.getExitString();
    /*System.out.print("Exits: ");


    if(currentRoom.getExit("north") != null) {
        System.out.print("north");
    }

    if(currentRoom.getExit("east") != null) {
        System.out.print("east");
    }
    if(currentRoom.getExit("west") != null) {
        System.out.print("west");
    }
    if(currentRoom.getExit("south") != null) {
        System.out.print("south");
    }
    */
    System.out.println();

    /*

    if(currentRoom.northExit != null) {
        System.out.print("north ");
    }
    if(currentRoom.eastExit != null) {
        System.out.print("east ");
    }
    if(currentRoom.southExit != null) {
        System.out.print("south ");
    }
    if(currentRoom.westExit != null) {
        System.out.print("west ");
    }

    */


}
```

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Implicit coupling

Linkage not necessarily expressed via data and method interactions.

Can arise from assumptions about the way a class is implementing.

Satisfactory resolution might require a small increase in explicit coupling.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

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.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

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.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Cohesion (reprise)

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).

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

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.

We aim to avoid loosely cohesive classes and methods:

- Methods performing multiple tasks.
- Classes modeling multiple entities.
- Classes having no clear identity.
- Modules/Packages of unrelated classes.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

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.

Module/Package level:

- Groups of related classes.

The classes

The **command words** class defines all the valid commands in the game. It holds an array of string objects representing the command words.

The **parser** reads lines of input from the terminal and tries to interpret them as commands. It creates objects of the class command that represent the command that was entered.

A **command** object represents a command that was entered by the user. It has methods to check whether this was a valid command and to get the first and second words of the command as separate strings.

A **room** object represents a location in the game. Rooms can have exits that lead to other rooms.

The **game** class is the main class of the game. It sets up the game and then enters a loop to read and execute commands. It also contains the code that implements each user command.

Ideate your game (work in teams)

P1. Design your own game scenario away from the computer. Don't worry about implementation or classes or programming. Just try and come up with something interesting. It needs to be the basic structure of a player moving through different locations. Possible examples:

- Finding the exit in a big shopping mall
- A mole must find the food hidden in one of his burrows before winter comes
- An adventurer is looking for a monster in a series of dungeons
- The bomb squad must find the room with the bomb before it goes off.
- The NSA is looking for Ed Snowdon and going from country to country....

Be creative! Give your game a name.

P2. What is the goal of your game, that is, when does the player win?

P3. What could you add to the game to make it interesting? Trap doors, treasure, monsters, ...

P4. Draw a map of your game layout.