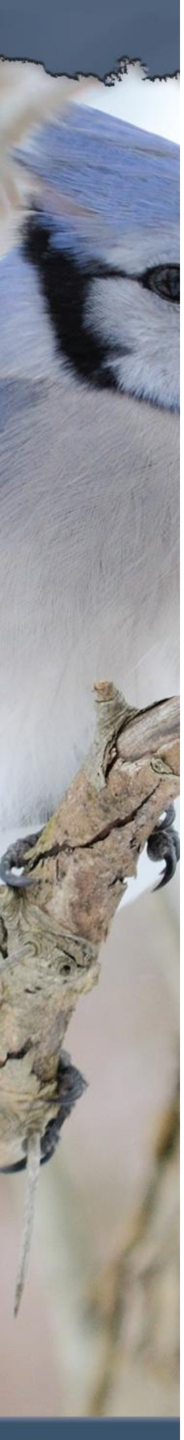
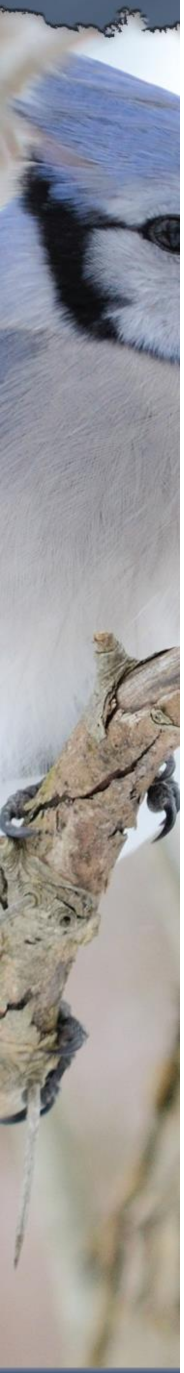


Designing classes

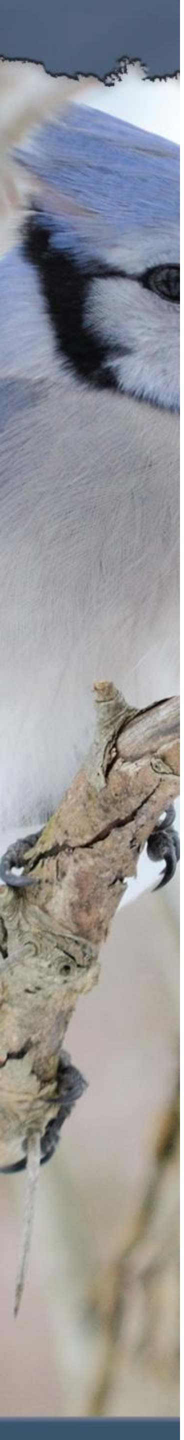
Easily understandable, maintainable and reusable Classes

Edited by Ehsan Edalat and Amir Kalbasi



Software changes

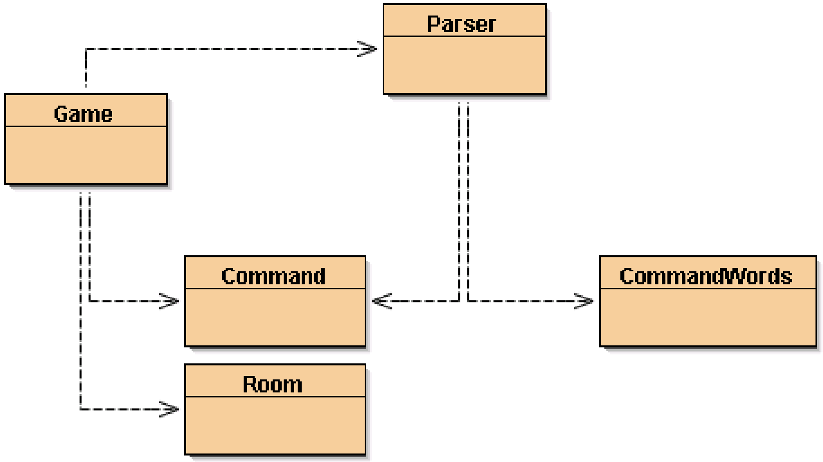
- Software is not like a novel that is written once and then remains unchanged
- Software is extended, corrected, maintained, ported, adapted, etc…
- The work is done by different people over time (often decades)



Change or die

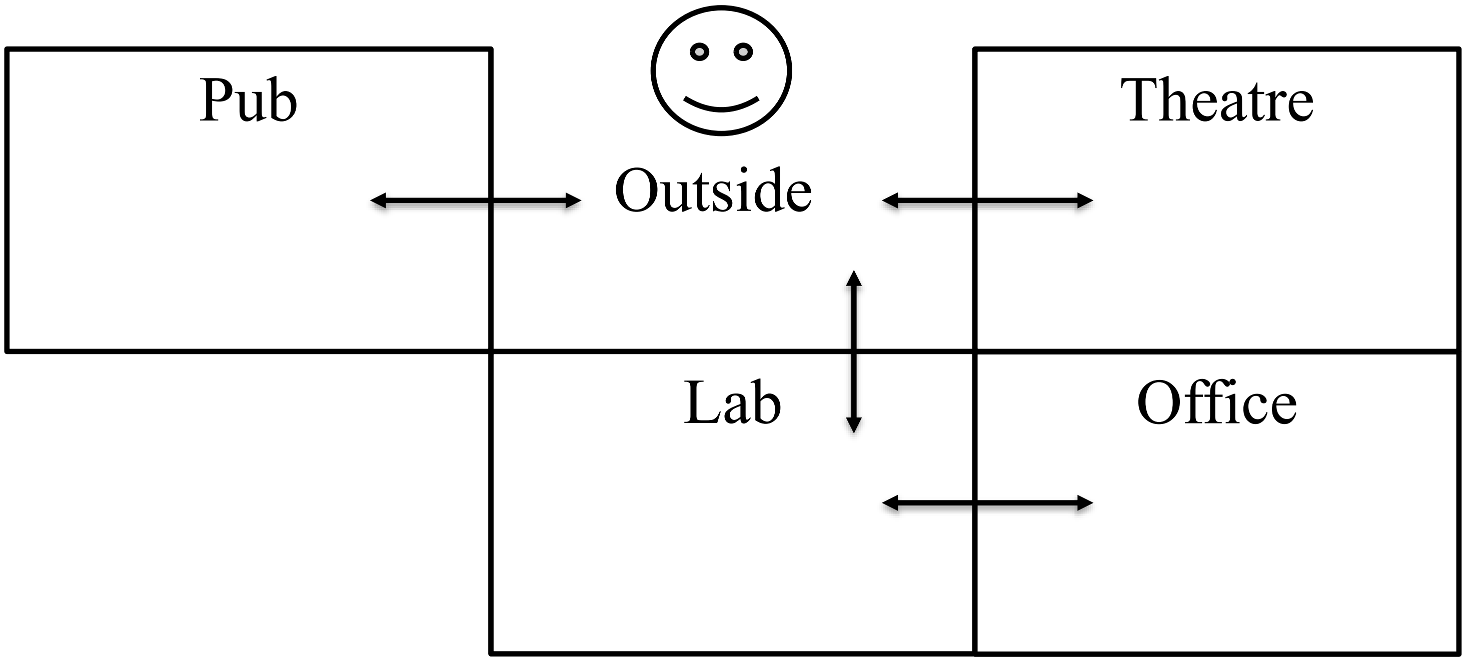
- There are only **two options** for software:
  - Either it is continuously maintained
  - or it dies
- Software that cannot be maintained will be thrown away

World of Zuul Classes

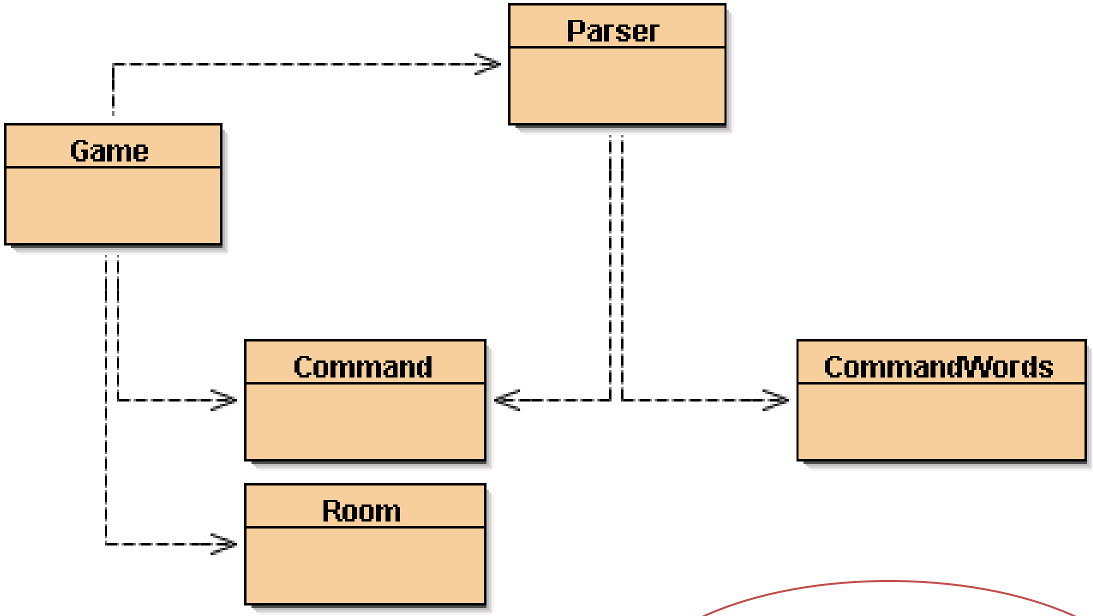


- **`Game`**: The starting point and main control loop
- **`Room`**: A room in the game
- **`Parser`**: Reads user input
- **`Command`**: A user command
- **`CommandWords`**: Recognized user commands

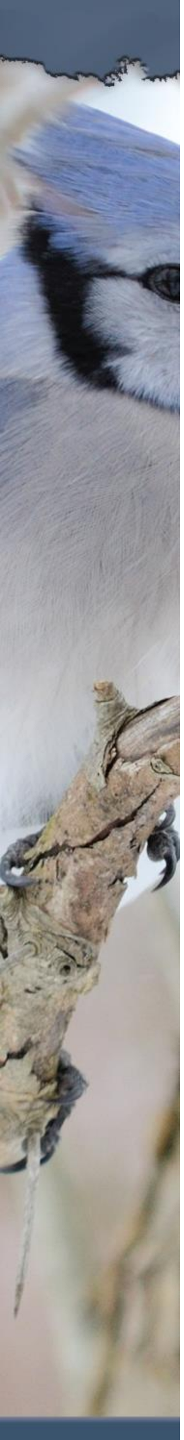
Designed Rooms



World of Zuul



Explore
`zuul-bad`

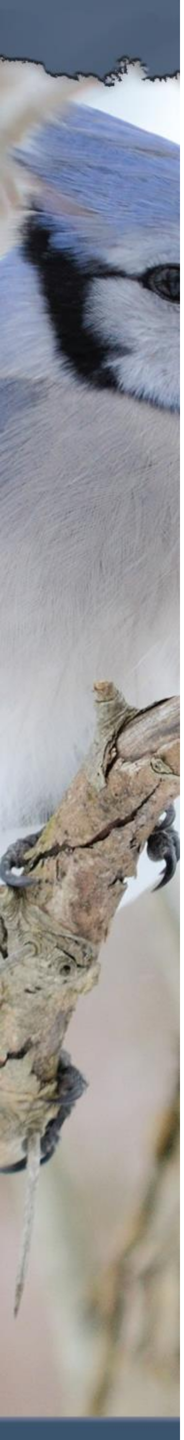


Code and design quality

- Criteria needed to define how to evaluate code quality
- Two important concepts for assessing the quality of code are:
  - Coupling
  - Cohesion

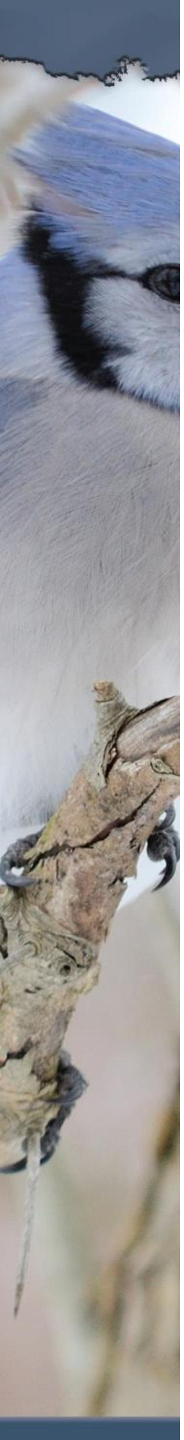
Coupling

- 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*
- However, we aim for *loose coupling*
  - where classes are not so inter-connected
- A class diagram provides hints at where coupling exists



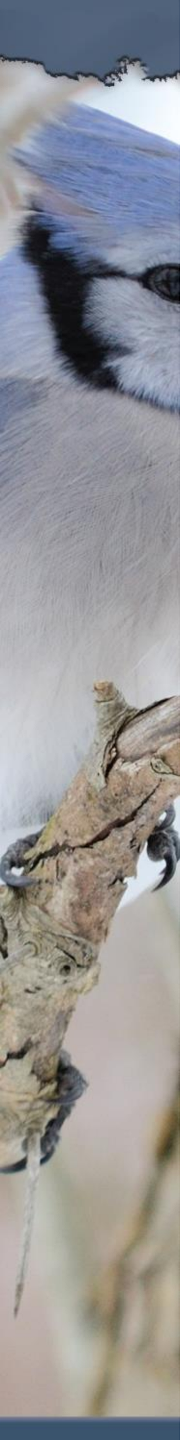
Loose coupling

- 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



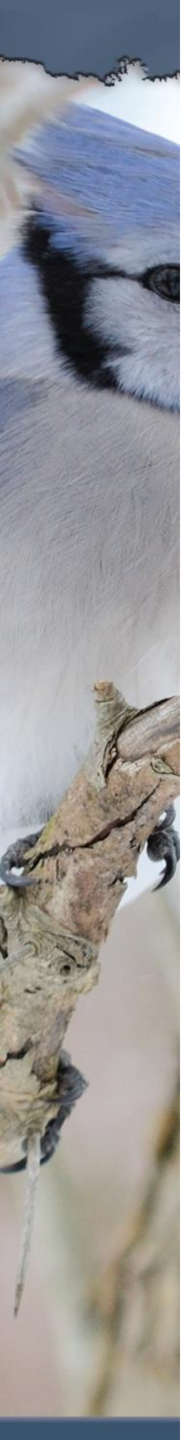
Tight coupling

- 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



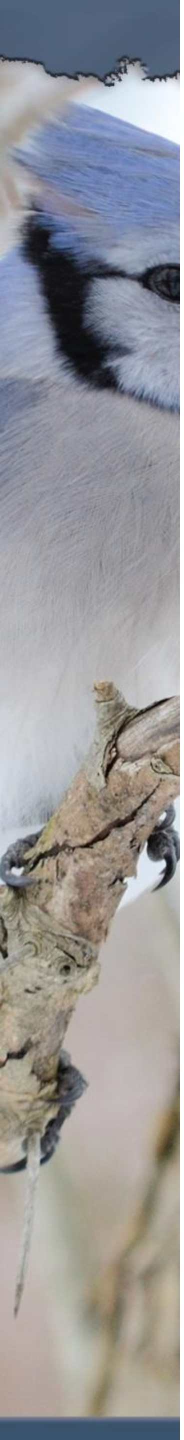
Cohesion

- 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
  - responsible for only one cohesive task
- A *unit* applies to classes, methods and modules (packages)
  - for reusability and maintainability



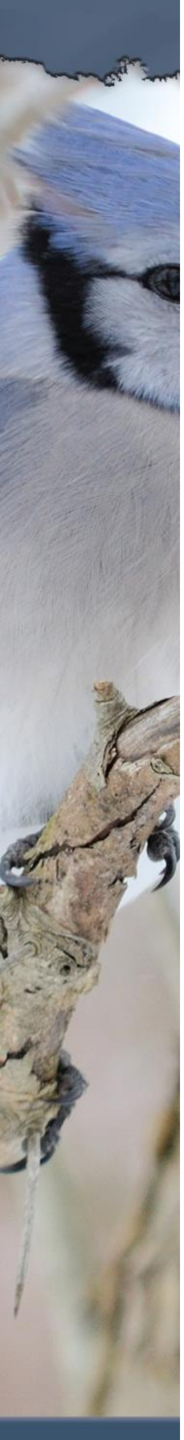
High cohesion

- 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
- Allows for readability and reuse



Loose cohesion

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

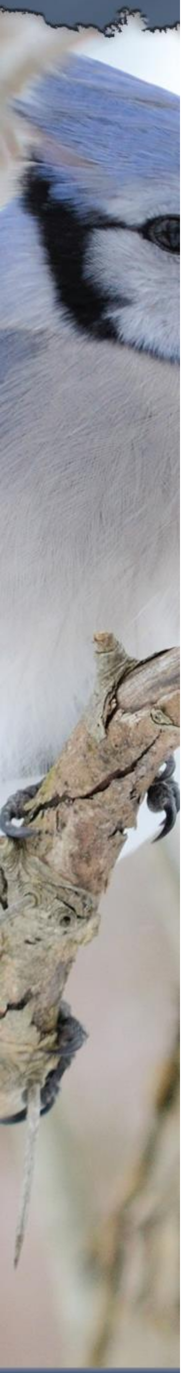


Cohesion applied at different levels

- 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
- Module/Package level:
  - Groups of related classes

An 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?



Finding relevant source code

- What do we change to add 2 new directions?

  Class *Room*

  - exits of each room stored as *fields*
  - exits assigned in *setExits* method

  Class *Game*

  - exit info printed in *printWelcome* method
  - exits defined in *createRoom* method
  - exits used in *goRoom* to find next room

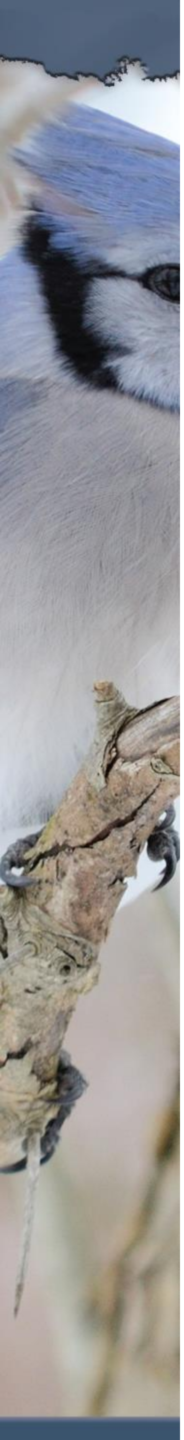
- Where and how easy is it to apply?

  Must add *up* and *down* options to ALL of these places … making it VERY difficult.

Encapsulation
to reduce coupling

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

What is wrong with the fields of this class *Room*?



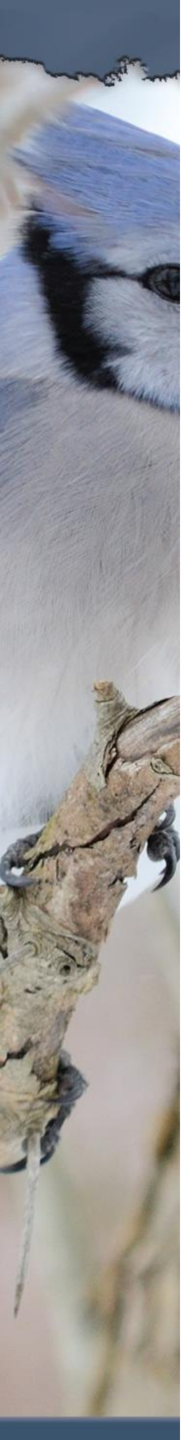
Encapsulation to reduce coupling

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

What is wrong with the fields of this class *Room*?

Fields are declared as <u>public</u>!!

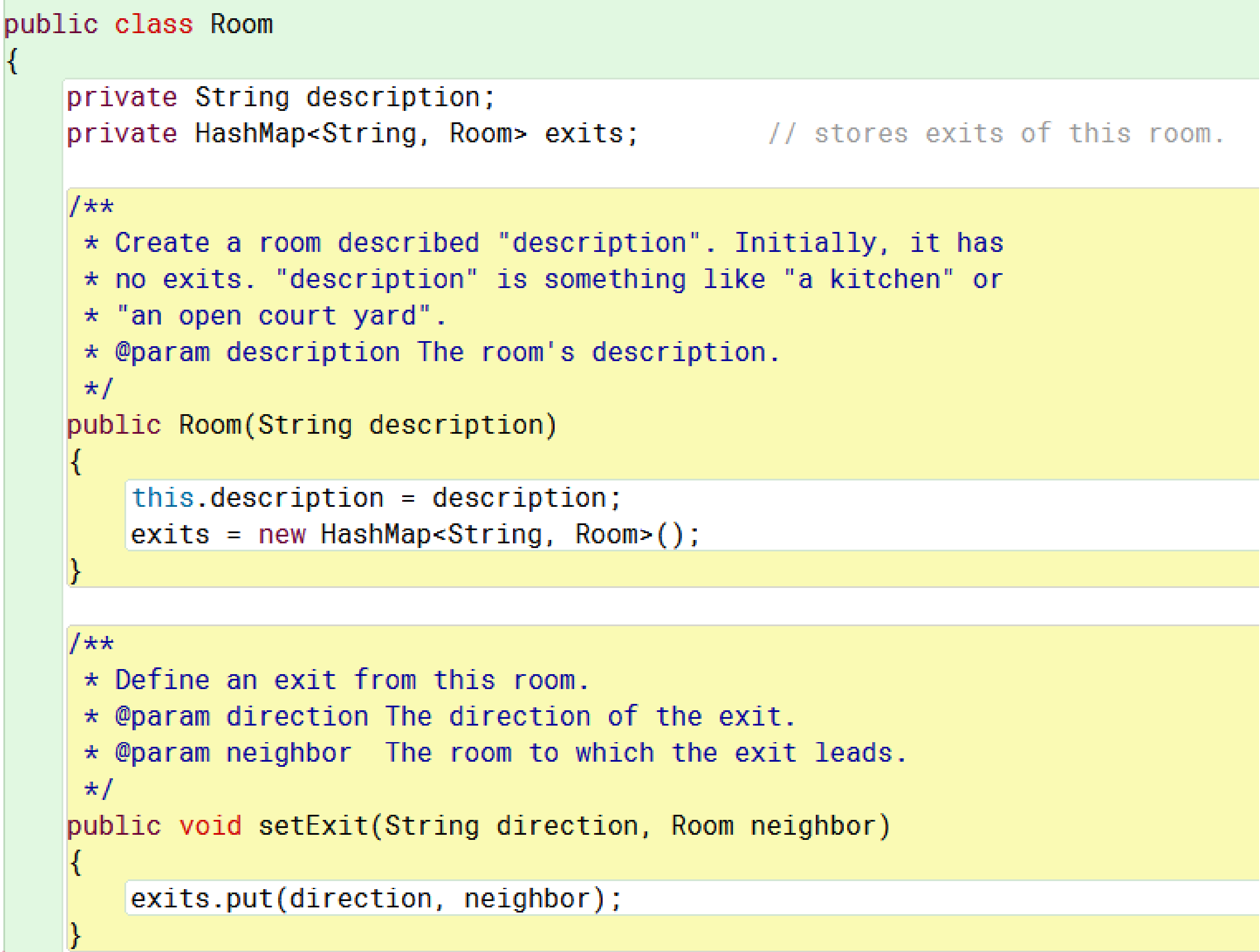
- allows direct access from ANY other class
- exposes <u>*how*</u> exit information is stored
- no longer hides *implementation* from view
- breaks encapsulation guideline suggesting only <u>*what*</u> a class does is visible to the outside



Reducing coupling

- Encapsulation supports loose coupling
  - private elements cannot be referenced from outside the class
  - Reduces the impact of internal changes

Changing the type of storing data in Room class



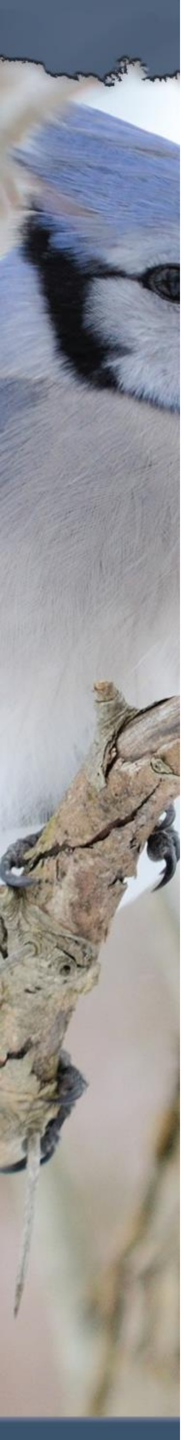
```
public class Room
{
    private String description;
    private HashMap<String, Room> exits;        // stores exits of this room.

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

    /**
     * Define an exit from this room.
     * @param direction The direction of the exit.
     * @param neighbor  The room to which the exit leads.
     */
    public void setExit(String direction, Room neighbor)
    {
        exits.put(direction, neighbor);
    }
```

Code duplication

(Loose cohesion)

Both the *printWelcome* & *goRoom* methods contain the following lines of code to print the current room details:

```
System.out.println("You are " +
                        currentRoom.getDescription());
System.out.print("Exits: ");
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 ");
}
System.out.println();
```

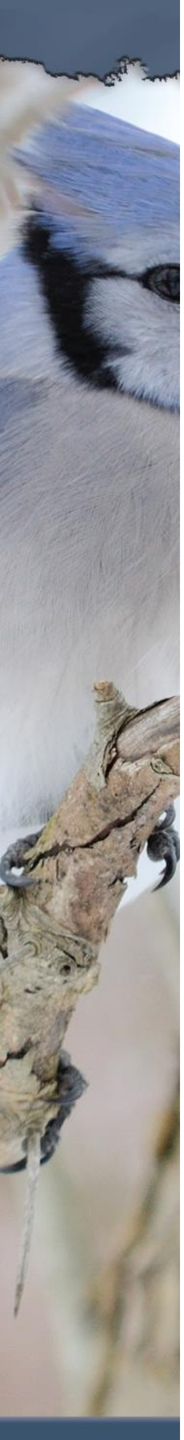


Avoid code duplication for high cohesion

- Code duplication
  - is an indicator of bad design
  - makes maintenance harder
  - increases chance of inconsistencies
  - leads to errors during maintenance
  - not all copies of code are changed
  - *loose cohesion* with parts of multiple method doing the same thing
  - separate into more cohesive units

printLocationInfo( )

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

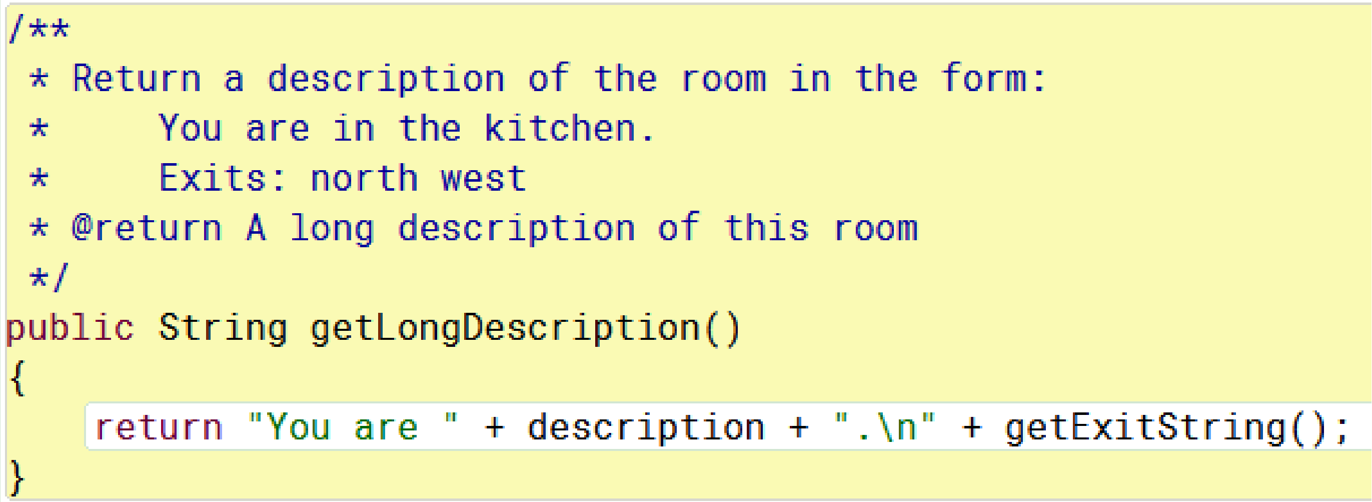


Responsibility-driven design

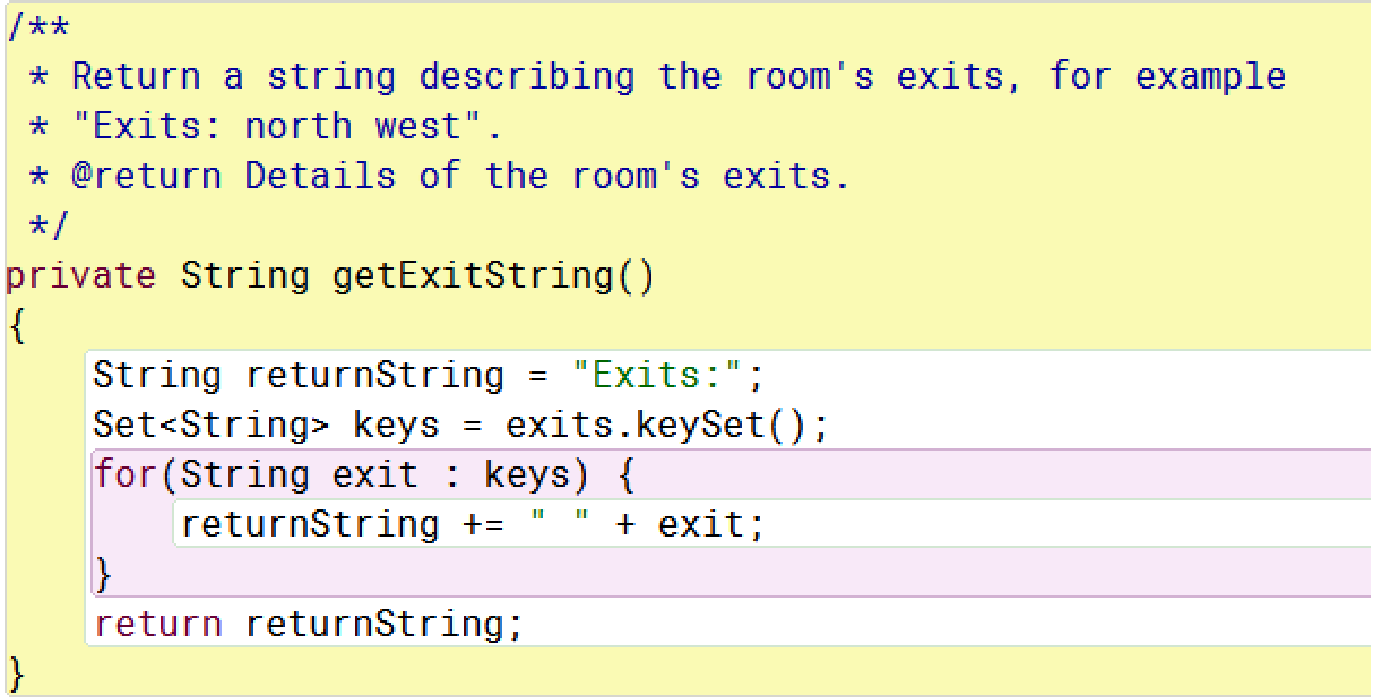
Where should we add a new method (which class)?

- 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

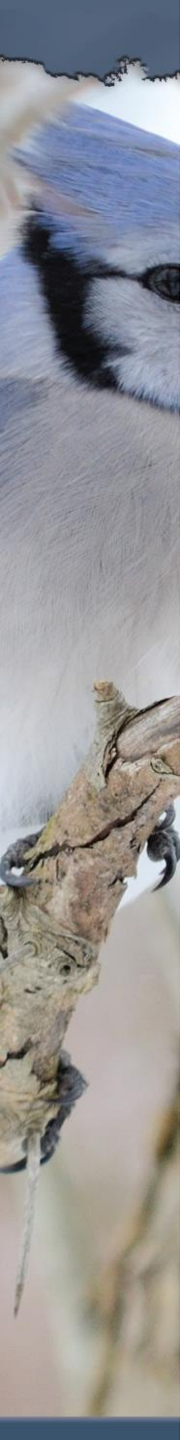
Responsibility-driven design



```
/**
 * Return a description of the room in the form:
 *      You are in the kitchen.
 *      Exits: north west
 * @return A long description of this room
 */
public String getLongDescription()
{
    return "You are " + description + ".\n" + getExitString();
}
```



```
/**
 * Return a string describing the room's exits, for example
 * "Exits: north west".
 * @return Details of the room's exits.
 */
private String getExitString()
{
    String returnString = "Exits:";
    Set<String> keys = exits.keySet();
    for(String exit : keys) {
        returnString += " " + exit;
    }
    return returnString;
}
```



Localizing change

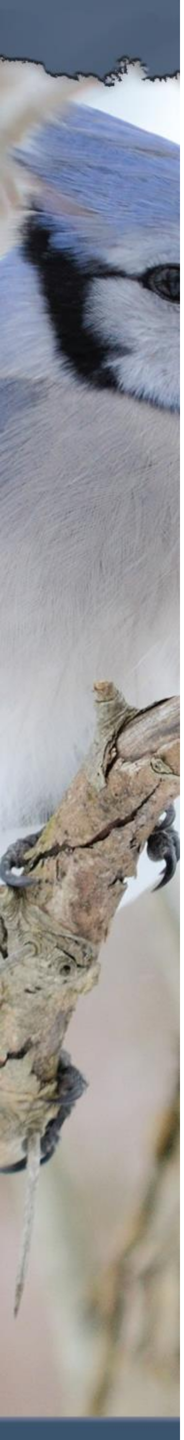
- One aim of reducing coupling and responsibility-driven design is to localize change
- When a change is needed, as few classes as possible should be affected

Thinking ahead

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

Suppose an existing program is upgraded from a textal interface to graphical:

- Replace ALL System.out.println statements
- Too many *hard-coded* instances to change
- Better to *encapsulate* all information about the user interface in a single class … at the start
- Then other classes should *produce* information to pass to the *"user interface"* class to present
- So changes to the user interface would be localized to only 1 class … the *"user interface"*



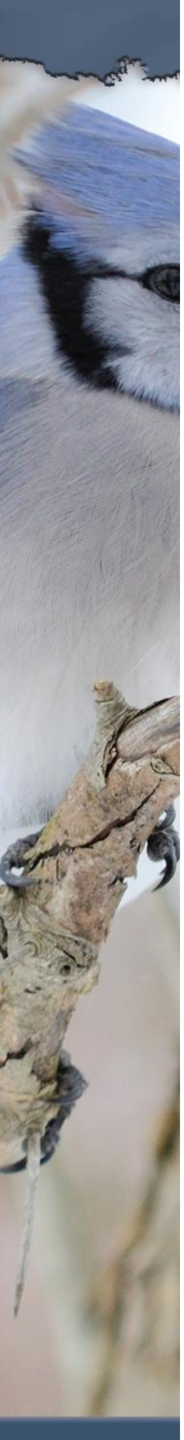
Refactoring

- When classes are maintained or changed, often new code is added
- Classes and methods tend to become longer, possibly losing high cohesion and loose coupling
- Every now and then, classes and methods should be *refactored* to maintain its high cohesion and low coupling
- Refactoring means rethinking and redesigning the program’s class and method structures

Refactoring and testing

HOWEVER ...

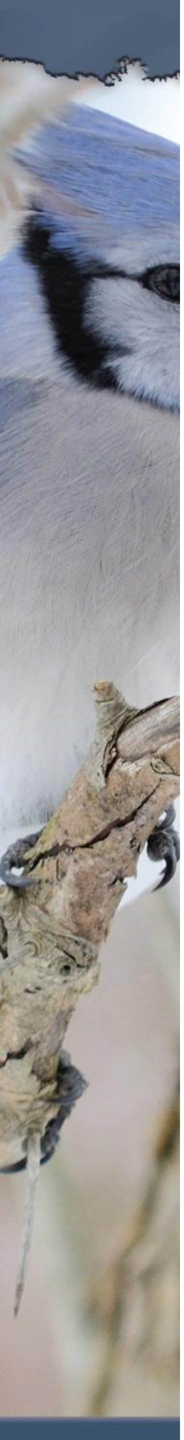
- When refactoring code, separate the refactoring from making other changes
- First, do the refactoring ONLY without changing the functionality
- Test before and after refactoring to ensure that nothing was broken
- Then, continue with maintenance or changes



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

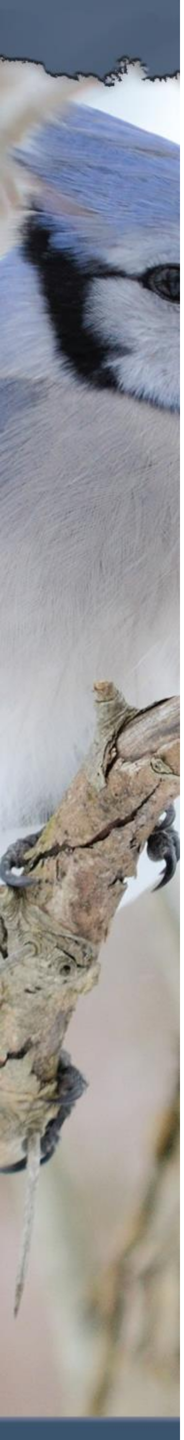
How complex should a class be?

- A class is too complex if it represents more than one logical entity

How long should a method be?

- A method is too long if it does more then one logical task

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



Enumerated Types

- A language feature defining a type
- Declared like a class using ***enum*** instead of ***class*** to introduce a type name
- Used to define a list of variable names denoting the set of values belonging to this type:
  - Alternative to static ***int*** constants
  - When the constants’ values would be arbitrary

A basic enumerated type

```
public enum CommandWord
{
    GO, QUIT, HELP, UNKNOWN
}
```

- By convention, names are defined in CAPS
- 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

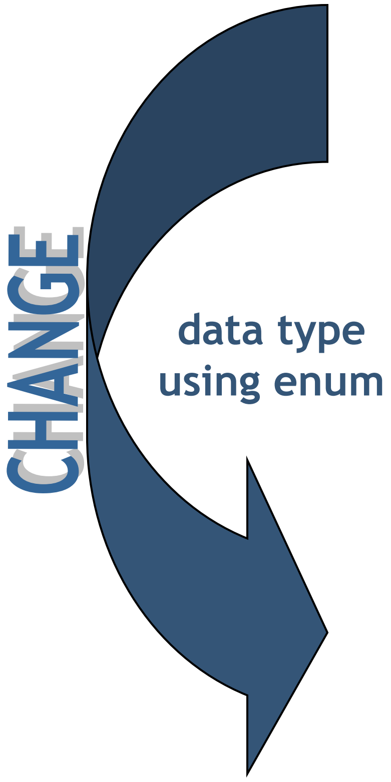
Using enumerated types

```
public enum CommandWord
{
    GO, QUIT, HELP, UNKNOWN
}
```

```
String commandWord = command.getCommandWord();
if(commandWord.equals("help")) {
      printHelp();
}
else if(commandWord.equals("go")) {
      goRoom(command);
}
else if(commandWord.equals("quit")) {
      wantToQuit = quit(command);
}
```

String type
commandWord

```
if(commandWord.equals("help")) {
        printHelp();
}
else if(commandWord.equals("go")) {
        goRoom(command);
}
else if(commandWord.equals("quit")) {
        wantToQuit = quit(command);
}
```



CHANGE

data type
using enum

```
public enum CommandWord
{
     GO, QUIT, HELP, UNKNOWN
}
```

CommandWord type
commandWord

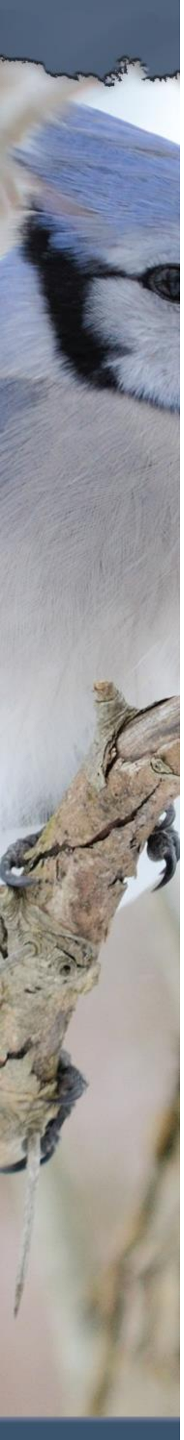
```
if(commandWord == CommandWord.HELP) {
        printHelp();
}
else if(commandWord == CommandWord.GO) {
        goRoom(command);
}
else if(commandWord == CommandWord.QUIT) {
        wantToQuit = quit(command);
}
```

```
if(commandWord == CommandWord.HELP) {
        printHelp();
}
else if(commandWord == CommandWord.GO) {
        goRoom(command);
}
else if(commandWord == CommandWord.QUIT) {
        wantToQuit = quit(command);
}
```

BEST

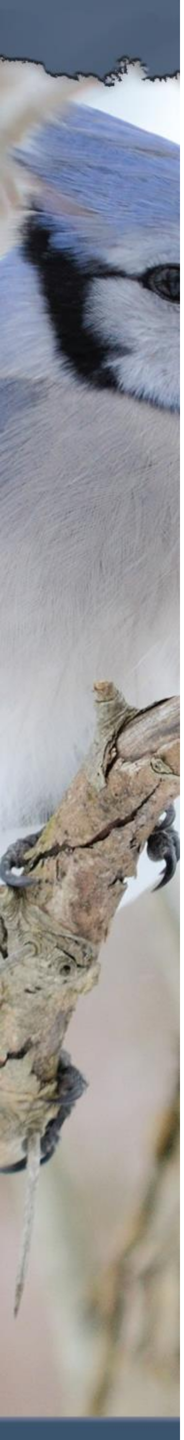
Use *switch* to express code intent even more clearly …

```
switch (commandWord) {
        case HELP:
                printHelp();
                break;
        case GO:
                goRoom(command);
                break;
        case QUIT:
                wantToQuit = quit(command);
                break;
}
```



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 very important
- There is a big difference in the amount of work required to change poorly-structured and well-structured code … so make your code count!!