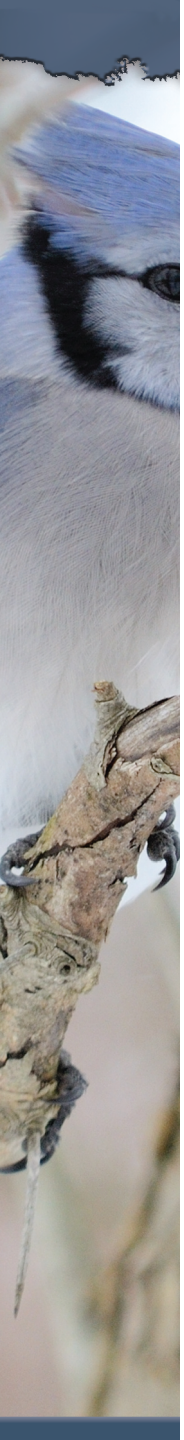


Designing classes

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

"Though a program be but three lines long, someday it will have to be maintained." -- The Tao of Programming

Main concepts to be covered

- Responsibility-driven design
- Coupling
- Cohesion
- Refactoring

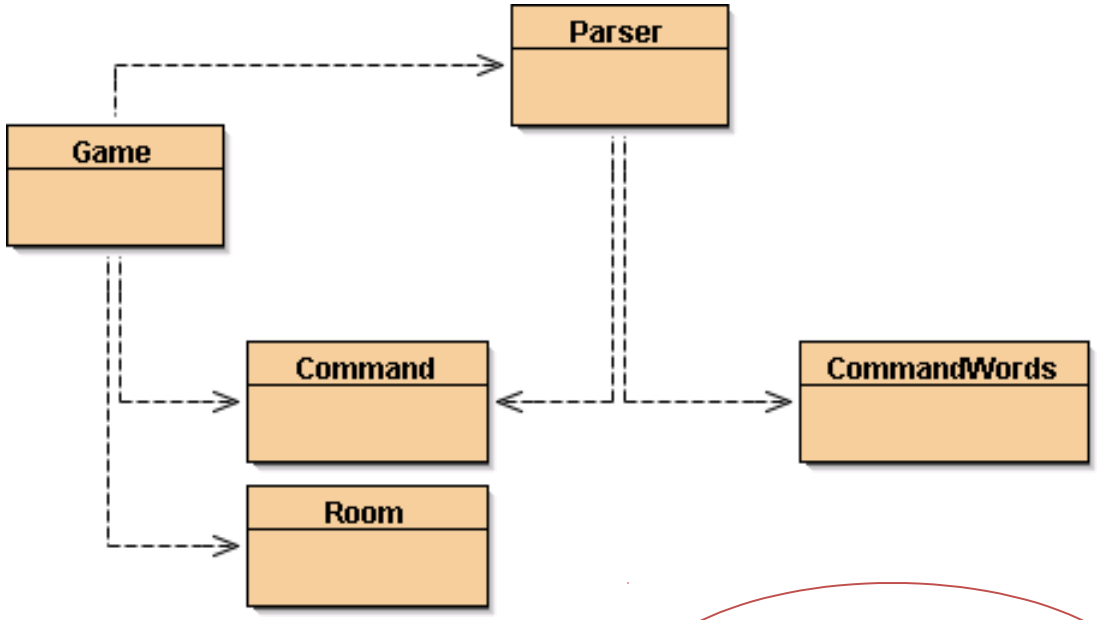
Software changes

- 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.

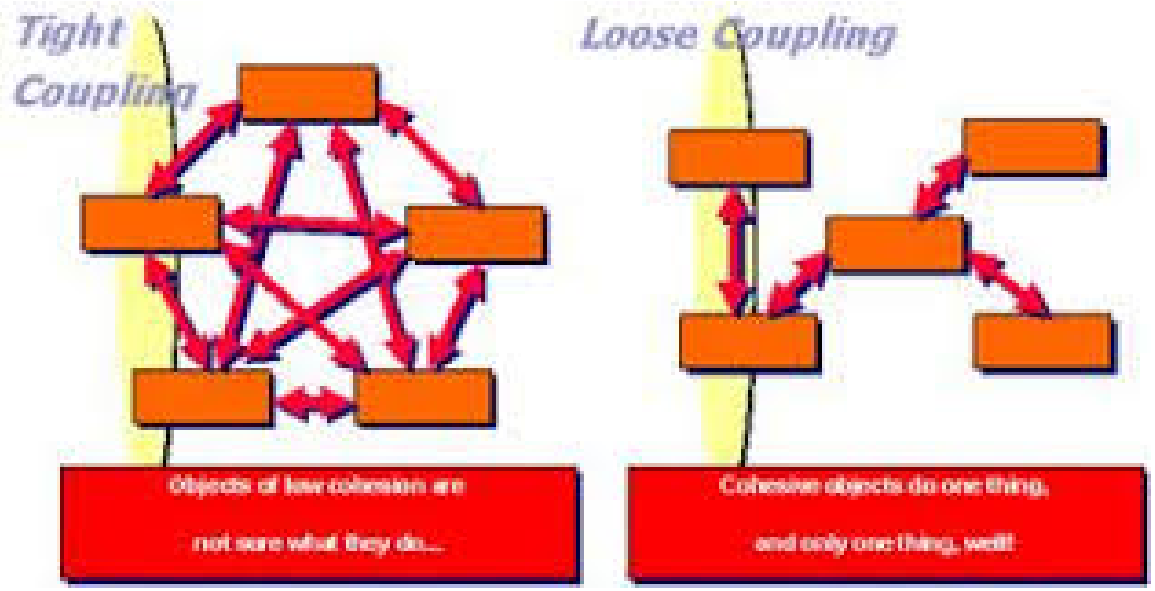
Code and design quality

- 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 hints at where coupling exists.

Coupling


Tight Coupling
Loose Coupling
Objects of low cohesion are
not sure what they do...
Cohesive objects do one thing,
and only one thing, well!

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).

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?

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

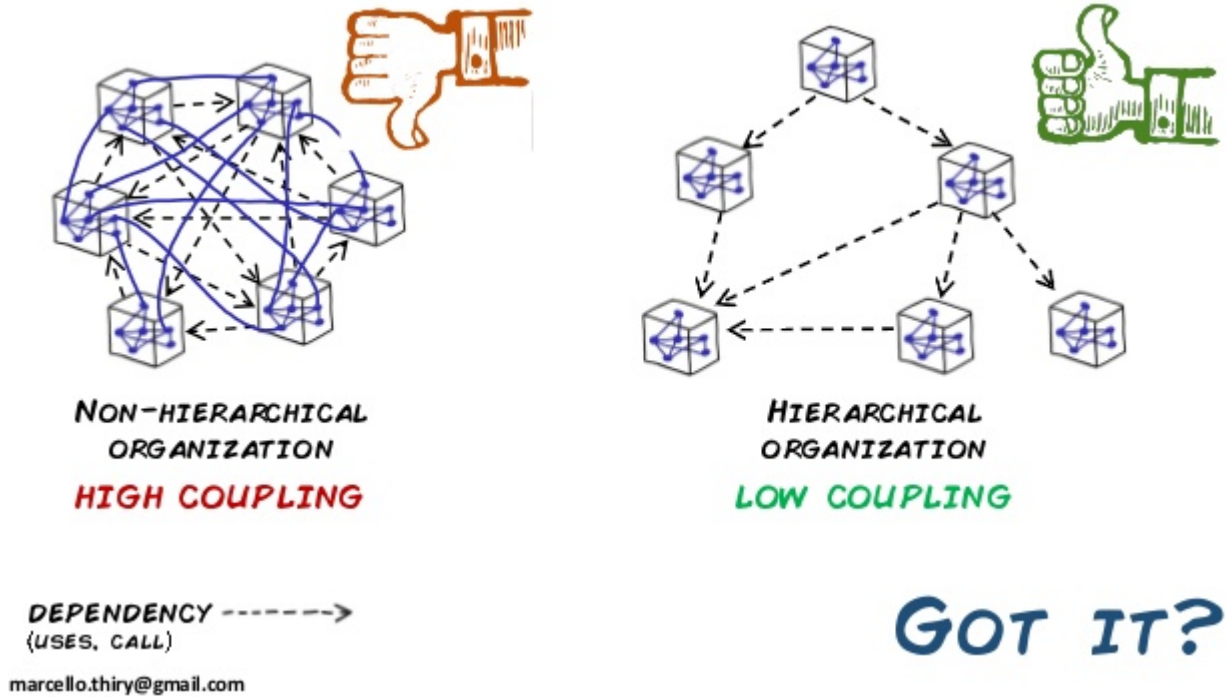
- We try to avoid tight coupling.
- Changes to one class bring a cascade of changes to other classes.
- Classes are harder to understand in isolation.
- Flow of control between objects of different classes is complex.

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.

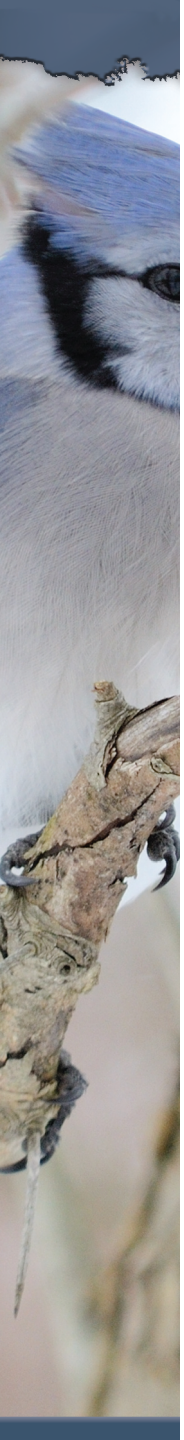
Reducing coupling

- Encapsulation supports loose coupling.



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.



RDD example

- **`SomeClass#getNumber()`** produces a number.
- You realise you often get the number and compute its √.
- You should introduce a new method **`SomeClass#getSqrtNumber()`**.
- **`SomeClass`** now has responsibility for manipulating its data.

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.

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).

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.

Loose cohesion

- 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.

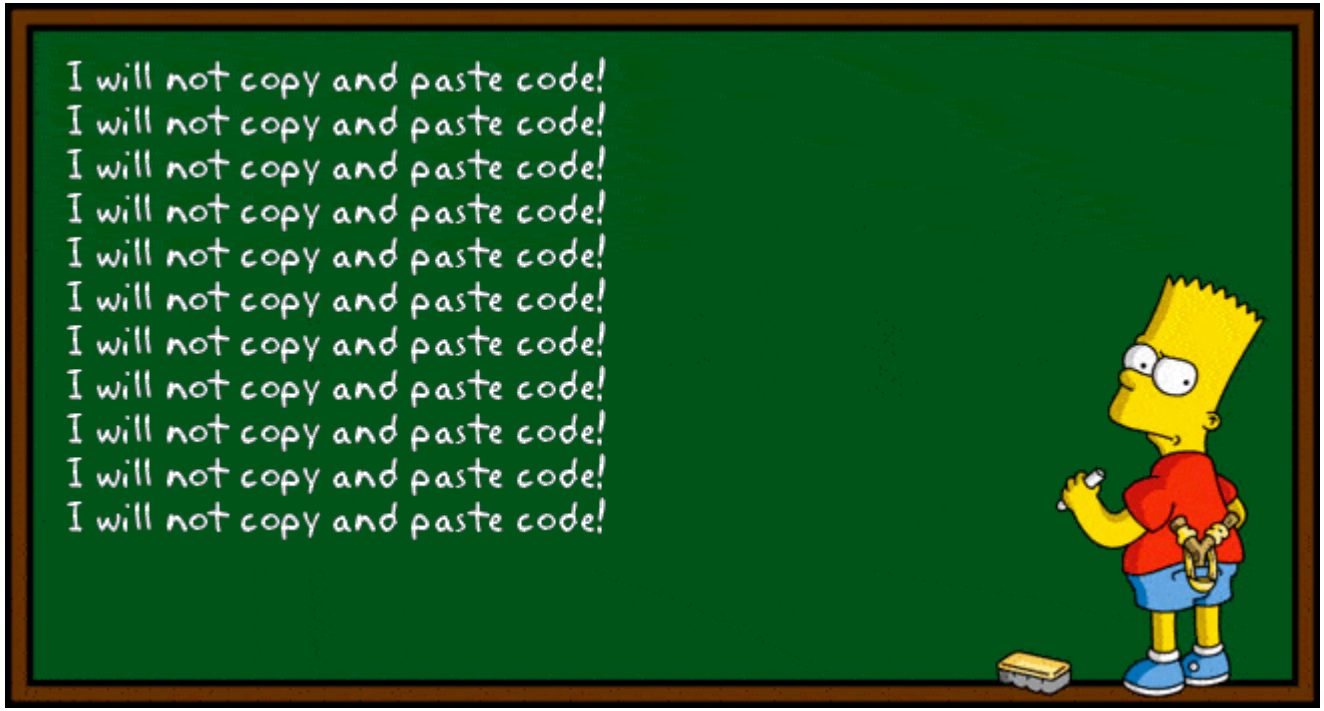
Cohesion applied at different levels

- 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.

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.

Code duplication is baaaad

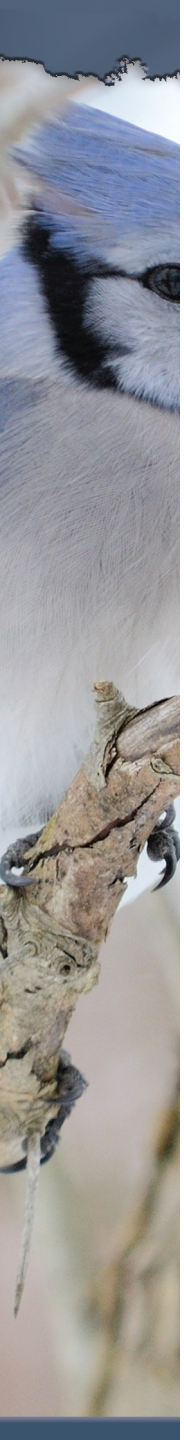


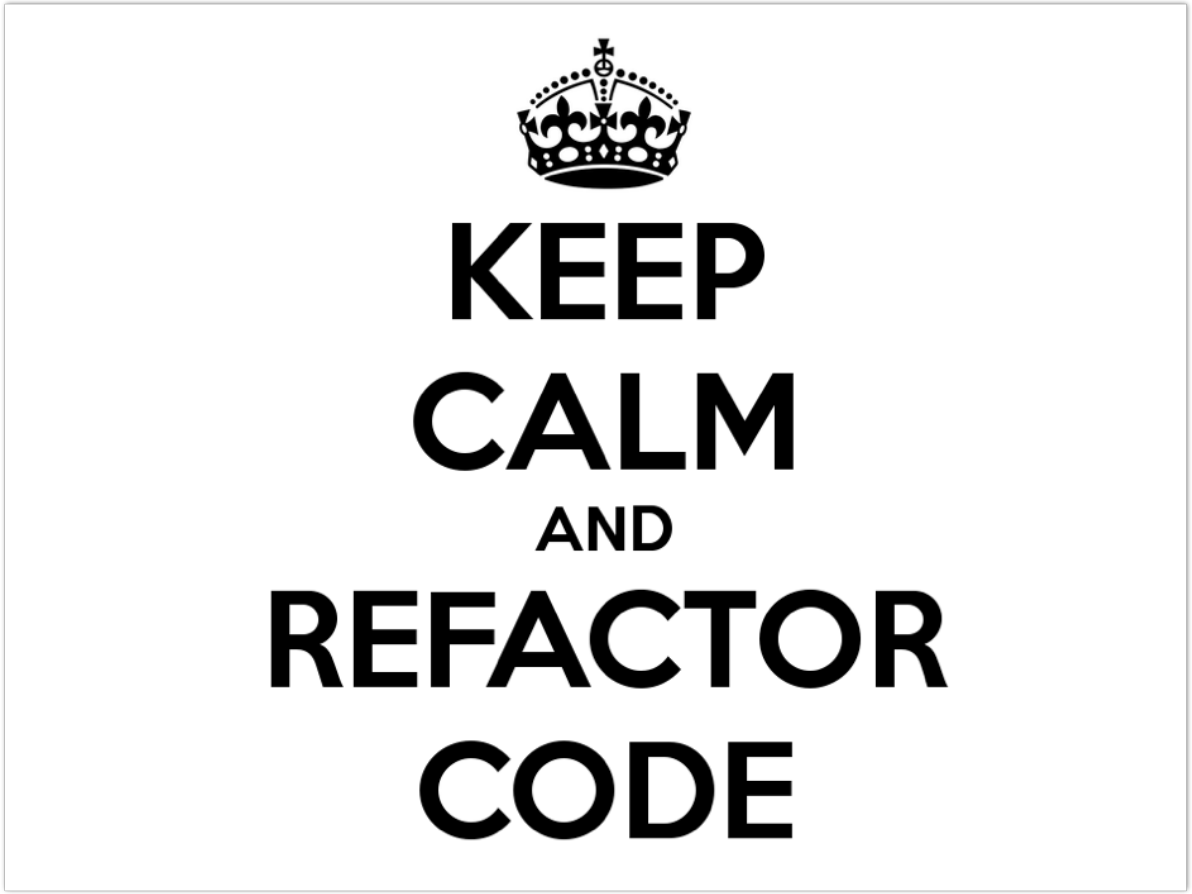
Thinking ahead

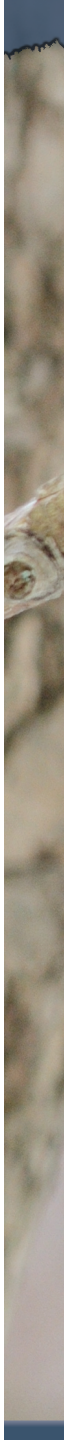
- When designing a class, we try to think about changes likely to be made in the future.
- We aim to make those changes easy.
- Requires a little more effort now to greatly reduce effort later.

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.



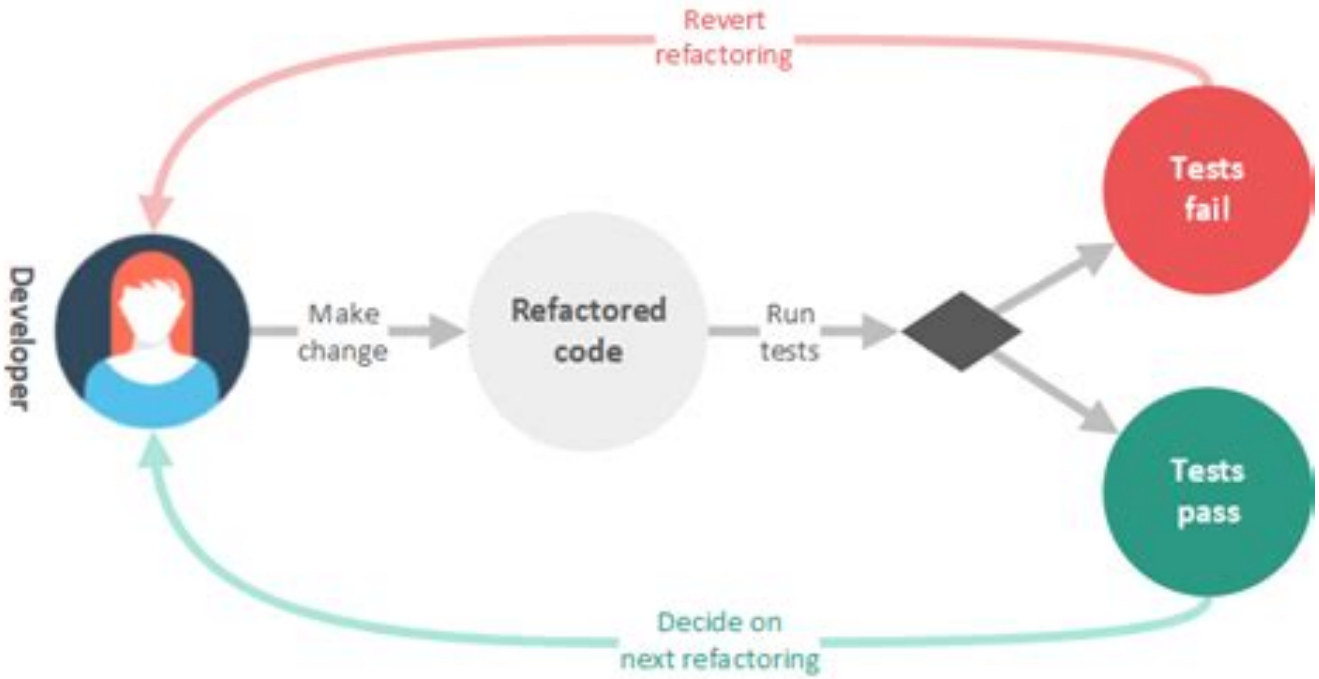

KEEP
CALM
AND
REFACTOR
CODE



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.

Refactoring and testing



Design questions

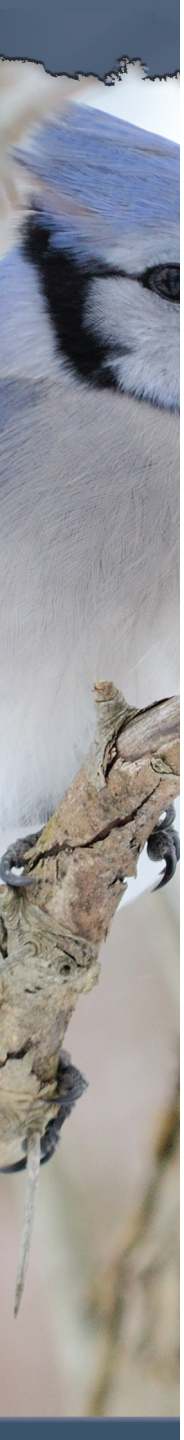
- Common questions:
  - How long should a class be?
  - How long should a method be?

- These can now be answered in terms of cohesion and coupling.

Design guidelines

- A method is too long if it does more 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.



Assuming responsibility

- Access should be just enough to get the job done, but no more.
- Why not make attributes public?

```
public class Car {
    public int speed;

    // code using speed
}
```

Assuming responsibility

- Access should be just enough to get the job done, but no more.
- Why not make attributes public?

```
public class Car {
    public int speed;

    // code using speed
}
```

```
// probably don't want this
Car car = new Car();
car.speed = 210;  // bad idea
```

Assuming responsibility

- Class assumes responsibility:

```
public class Car {
    private int speed;
    public void setSpeed(int newSpeed) {
        speed = newSpeed < MAX_SPEED
                ? newSpeed : MAX_SPEED;
    }
    // code using speed
}
```

Assuming responsibility

- Class assumes responsibility:

```
public class Car {
    private int speed;
    public void setSpeed(int newSpeed) {
        speed = newSpeed < MAX_SPEED
                ? newSpeed : MAX_SPEED;
    }
    // code using speed
}
```

```
// evil code fails to be evil
Car car = new Car();
car.speed = 210;  // nope - compiler error
car.setSpeed(210);  // ok - limited speed
```

Assuming responsibility

- However, **`setSpeed`** supposes infinite acceleration

Assuming responsibility

- Class assumes more responsibility:

```
public class Car {
    private int speed;
    private void setSpeed(int newSpeed) {
        speed = newSpeed < MAX_SPEED
                ? newSpeed : MAX_SPEED;
    }
    public void accelerate(int newSpeed) {
        // take time to get to newSpeed
        setSpeed(newSpeed);
    }
    // code using speed
}
```

Assuming responsibility

- Class assumes more responsibility:

```
public class Car {
    private int speed;
    private void setSpeed(int newSpeed) {
        speed = newSpeed < MAX_SPEED
                ? newSpeed : MAX_SPEED;
    }
    public void accelerate(int newSpeed) {
        // take time to get to newSpeed
        setSpeed(newSpeed);
    }
    // code using speed
}
```

```
// evil code fails to be evil
Car car = new Car();
car.setSpeed(210);  // compiler error
car.accelerate(210);  // not instantaneous
```

Assuming responsibility

- Beware of getters:

```
public class Car {
    private final Map<String, String> parts
            = new HashMap<>();
    public Car() {
        parts.put("engine", "big block V8");
    }
    public Map<String, String> getParts() {
        return parts;
    }
}
```

Assuming responsibility

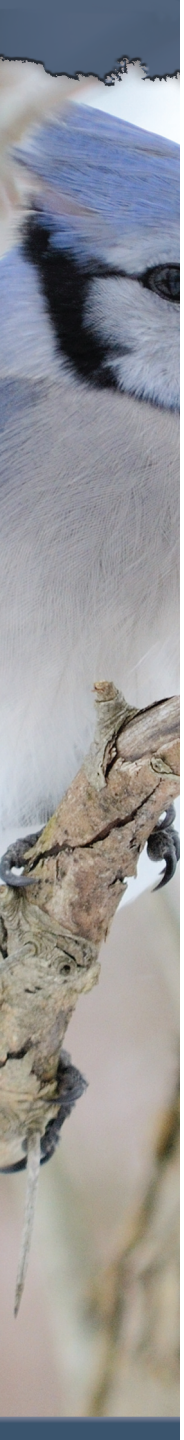
- Beware of getters:

```
public class Car {
    private final Map<String, String> parts
            = new HashMap<>();
    public Car() {
        parts.put("engine", "big block V8");
    }
    public Map<String, String> getParts() {
        return parts;
    }
}
```

```
// evil garage code
Car car = new Car();
Map<String, String> parts = car.getParts();
parts.put("engine", "tiny 3 cylinder");
// dude, who stole my engine?!!??
```

Paranoia summary

- Allow minimal access to your code.
  - Start with everything **`private`**.
  - Enlarge to package-private if necessary.
  - Enlarge to **`protected`** only if necessary.
  - Enlarge to **`public`** only if really necessary.
- Use **`final`** as much as possible.
- Don't return references to collection attributes.
- Be very careful returning references to mutable objects.



Paranoia summary

- Try not to accept / use references to mutable objects (including arrays and collections).
- Create *defensive copies*:
  - When receiving mutable objects.
  - When returning mutable objects.

Enumerated Types

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

A basic enumerated type

```
enum CommandWord {
    // A value for each command word,
    // plus one for unrecognised 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.

A less basic enumerated type

```
enum CommandWord {
    GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("?");
    private String commandString;
    CommandWord(String commandString) {
        this.commandString = commandString;
    }
    public String toString() {
        return commandString;
    }
}
```

- An `enum` is like a `class`:
  - instance variables
  - constructor (always private)
  - methods

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.