B6582784

Generally, In order to achieve robustness( and high cohesion), programmers follow a certain set of rules and principles for better organization of object-oriented programming

Single Responsibility Principle

Section 3, Lecture 37

**Single Responsibility Principle**

Object-Oriented Terminology

In object-oriented programming (Java, among other languages, follows this paradigm), you will often hear terms such as robustness, cohesion, coupling etc. [**Cohesion**](https://en.wikipedia.org/wiki/Cohesion_%28computer_science%29) is a way to measure how much the code segments within one module (methods of a class, classes inside a package…) belong together. The higher the cohesion – the better, since high cohesion implies easier maintenance and debugging, greater code functionality and reusability. The term cohesion is sometimes contrasted with the concept of [**coupling**](https://en.wikipedia.org/wiki/Coupling_%28computer_programming%29), and often, loose coupling of modules is related to high cohesion.

Another widely used term is **robustness,**which could be defined as the ability of a computer system or algorithm to handle mistakes and malfunctions (which could be caused by various factors such as programmer’s mistake or incorrectly formatted user input). A robust system is one that can handle these unwanted situations elegantly. There are various ways for a software engineer to achieve robustness, such as testing the code for different kinds of inputs, but generally, in order to achieve robustness (and high cohesion), programmers follow a certain set of rules and principles for better organization of object-oriented programs. One such principle is the single responsibility principle.

Single Responsibility Principle

The single responsibility principle revolves around the claim that a certain code module (most often, a class) should only have responsibility over one part of the functionality provided by the software. In software engineering books, this is sometimes also defined like this: the module should only have **one reason to change**. This means that a division of concerns is performed in the program, and the methods for every concern should be completely encapsulated by a single class. Now it is obvious that this approach contributes to the high cohesion – since methods related to the same concern (same part of the functionality) will be members of the same class, and robustness – since this reduces the possibility of error. Furthermore, if an error does occur, the programmer will be more likely to find the cause, and finally, solve the problem.

The single responsibility principle is founded on one of the basic, general ideas of object-oriented programming – the so-called *divide and conquer* principle – solving a problem by solving its multiple sub-problems. This approach prevents the creation of *“God objects”* – objects that “*know too much or do too much*“.

The classes you write, should not be a swiss army knife. They should do one thing, and to that one thing well.

(Bad) Example

Let’s consider this classic example in Java – “objects that can print themselves”.

1. class Text {
2. String text;
3. String author;
4. int length;
5. String getText() { ... }
6. void setText(String s) { ... }
7. String getAuthor() { ... }
8. void setAuthor(String s) { ... }
9. int getLength() { ... }
10. void setLength(int k) { ... }
11. /\*methods that change the text\*/
12. void allLettersToUpperCase() { ... }
13. void findSubTextAndDelete(String s) { ... }
14. /\*method for formatting output\*/
15. void printText() { ... }
16. }

At first glance, this class might look correctly written. However, it contradicts the single responsibility principle, in that it has multiple reasons to change: we have tho methods which change the text itself, and one which prints the text for the user. If any of these methods is called, the class will change. This is also not good because it mixes the logic of the class with the presentation.

Better Example

One way of fixing this is writing another class whose only concern is to print text. This way, we will separate the functional and the “cosmetic” parts of the class.

1. class Text {
2. String text;
3. String author;
4. int length;
5. String getText() { ... }
6. void setText(String s) { ... }
7. String getAuthor() { ... }
8. void setAuthor(String s) { ... }
9. int getLength() { ... }
10. void setLength(int k) { ... }
11. /\*methods that change the text\*/
12. void allLettersToUpperCase() { ... }
13. void findSubTextAndDelete(String s) { ... }
14. }
15. class Printer {
16. Text text;
17. Printer(Text t) {
18. this.text = t;
19. }
20. void printText() { ... }
21. }

Summary

In the second example we have divided the responsibilities of editing text and printing text between two classes. You can notice that, if an error occurred, the debugging would be easier, since it wouldn’t be that difficult to recognize where the mistake is. Also, there is less risk of accidentally introducing software bugs, since you’re modifying a smaller portion of code.

Even though it’s not that noticeable in this example (since it is small), this kind of approach allows you to see the “bigger picture” and not lose yourself in the code; it makes programs easier to upgrade and expand, without the classes being too extensive, and the code becoming confusing.

Single Responsibility Principle in Spring

As you become more comfortable using Spring components and coding to support Inversion of Control and [Dependency Injection in Spring](http://springframework.guru/dependency-injection-example-using-spring/), you will find your classes will naturally adhere to the single responsibility principle. A typical violation of the single responsibility principle I often see in legacy Spring applications is an abundance of code in controller actions. I’ve seen Spring controllers getting JDBC connections to make calls to the database. This is a clear violation of the single responsibility principle. Controller objects have no business interacting with the database. Nor do controllers have any business implementing other business logic. In practice your controller methods should be very simple and light. Database calls and other business logic belong in a service layer.