Good Code Design

Shannon Duvall

So far in your computing career you have probably been focused on making **functional** code. Your classes and assessments have been centered on the features of the software that are correctly implemented. That's pretty reasonable, since code that doesn't work is worthless.

Now that you are a good computer **programmer**, it's time to aspire to the next level: being a good computer **scientist**. The science of computing starts with making code that goes beyond functionality. This course is going to get you to think about two aspects of code you have not before: its efficiency and its design. Code that is merely functional is no longer good enough. Most of this course (and all of the textbook) is centered on learning how to analyze the efficiency of algorithms and create and implement your own efficient algorithms. This document, however, focuses on the other aspect of good code: code design.

What is good code design?

In the article "What Do Programmers Really Do Anyway?" Peter Hallam finds that he spends 2% of his time writing new code, 20% modifying existing code, and 78% of his time understanding existing code. Good code design is aimed at making the 98% of a coder's work (reading and modifying code) easier. Unlike in a university, in the real world programmers do not program by themselves, using a pre-defined design, on a completely new application. Rather, programming is done in teams, using existing code bases, with design of the code as important a task as the programming.

Code design is the choice of the structure and details of a program. It is making large-scale choices, like whether to use an interface or a superclass. It is making small-scale choices, like naming a variable. Each of these choices will make your code either easier or more difficult to read, understand, and modify later.

The aim of good code design is to make code that is:

- Easy to read and understand
- Easy to test
- Easy to reuse
- Easy to maintain, meaning:
    - Easy to extend functionality, without breaking existing code
    - Easy to debug

There are many books on code design and we could spend a whole course on it if we wanted. Instead, we are going to learn the very basics. I will use the vocabulary of Java and object-oriented programming, even though the principles I will highlight are valid in any language or programming paradigm.

Don't write STUPID code

The acronym STUPID tells us things to avoid in our code:

S = Singletons. Singletons are any object or variable that have only one instantiation. In general, we think of them as global variables, which means that any other piece of software can access it. Globals are undesirable because one piece of code can change a value and affect the functionality of code in some other place. When the code that causes a phenomenon is far from the code that shows the phenomenon, the code becomes difficult to understand and debug. This is colloquially called "spooky action at a distance."

T = Tight Coupling. Coupling is the degree to which two separate pieces of code are intertwined or dependent on one another. Tight coupling (as opposed to loose coupling) means that two pieces of code are very dependent on one another. This generally means one piece cannot be reused, modified, or tested without the other. Instead, each piece of code should have only one responsibility that stands alone as much as possible.

U = Untestable Code. This is a bit of an umbrella of several issues. Code becomes untestable when it uses globals, is tightly coupled with some other code, or does too much in one method.

P = Premature Optimization. Optimizing code is the process of making code more efficient in time, space, or both. Wait! I thought that was a good thing! Well, it is… to a certain extent. There are levels of optimization. At a high level the optimization has a large impact on performance. This generally involves choosing the algorithm and data structure that is most efficient (and that's what our course is about.) But lower levels of optimization (like using a switch statement instead of an if-else series) have very small effects on efficiency and almost always make the design of the code worse[1]. For example, consider this code:

```
foo(1);
bar(1);
baz(1);
foo(2);
bar(2);
baz(2);
foo(3);
bar(3);
baz(3);
```

You would probably never write code this way. Hopefully you would instead write:

```
for(int x = 1; x <=3; x++){
     doIt(x);
}

void doIt(int x){
     foo(x);
     bar(x);
     baz(x);
}
```

[1] No matter what Joel says. ☺

The two pieces of code do the same thing, but the first one has fewer variables and method calls and therefore is more efficient. Why then do we prefer the second way? It is easier to read and debug. Think about modifying the code from iterating 1-3 to 1-5. What would you need to change in each example? How likely are you to make a bug?

In general, low levels of optimization can be done automatically, when necessary, after the code has been written with good design. The "father of algorithm analysis" Don Knuth wrote, "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

I = Indescriptive Naming. You should always use naming and capitalization conventions, and be consistent. Favor clarity over brevity. It's okay if variable names are a bit long if it makes clear what data they represent. This may seem obvious, but it is actually more difficult than you'd think to devise clear and concise names.

D = Duplicated Code. If you have had a databases class, you have probably learned that duplicated data causes confusion and problems. It uses more space and takes more effort to change. But the real problem it creates is when one copy is updated and the other is not. At that point you cannot tell which copy is correct. Duplicated code in any form (not just data) makes code harder to reuse, debug, and change.

Write SOLID Code

The opposite of STUPID code is SOLID code. The SOLID acronym tells the hallmarks of good code. Since some of the design principles are more involved, we are only going to focus on the first two:

S = Single Responsibility Principle. A piece of code should do one and only one thing. If an object has too much responsibility, separate it into two. If a method is long, break it up. When a piece of code does only one thing, it is easier to understand, test, debug, and reuse.

O = Open/Closed Principle. The open/closed principle is the most important software principle. The others in the SOLID acronym are subsumed by the open/closed principle. This principle states that code should be **open to extension and closed to modification.** A piece of code is open to extension if it is written in such a way that adding functionality is easy. This generally means the inheritance hierarchies are in place so that adding new functionality means adding new subclasses. A piece of code is closed to modification if existing code does not have to be changed to add the functionality. Remember that any code that changes opens itself up to new bugs and therefore has to be re-tested. The goal is to make stable code that doesn't need changes to evolve. When changes do need to be made, we prefer the change to be to a variable or data structure than in logical code. The logic of the code is generally where mistakes are made.

We will not go over the rest of the acronym, but please look it up if you are interested:

L = Liskov Substitution Principle
I = Interface Substitution Principle
D = Dependency Inversion Principle

These principles are more detailed ways of making code that follows the open/closed principle.

OO Structure

The larger design questions in the object-oriented world center on how inheritance hierarchies and interfaces are defined. Here are a few rules of thumb to keep in mind:

- An inheritance structure defines an *is-a* relationship. (A BookStack is a Stack)
- Aggregation (having one object inside another) defines a *has-a* relationship. (A BookStack has a Book)
- Interfaces define an *acts-as-a*[2] relationship. (A BookStack acts as a Sortable)

These can help you use your intuition to know if something should be a superclass, interface, or container for another. When I was working as an interviewer, one of my favorite Java questions was "What is the difference between an abstract superclass and an interface?" The high-level answer is that they define two different types of relationships.[3]

Code Smells

If I were to ask any of you whether or not you want to write testable, maintainable code, I'm sure you would all enthusiastically say yes. So far, however, the information has been pretty academic. Let's get to the details – how will I know if my code is SOLID or STUPID? (How will my professor know?) The answer is to look for "code smells" – a construct in your code that points to a design flaw. Code smells are not bugs, they are symptoms of violating the principles we have talked about. Just as "debugging" is removing bugs from your code, the process of removing code smells and other poor design choices is called "refactoring". Refactoring improves the design of code without changing its functionality. Here are some common code smells:

- Duplicated code. If you ever have the urge to copy-and-paste while coding, stop and think about design.
- Large classes or methods. Make sure you follow the single responsibility principle.
- Inappropriate intimacy/ Feature Envy. This is when one class depends on the details of another class. Look for code that has a lot of get() calls on another object. This is a symptom of tight coupling.
- Freeloader. A class or method that does too little.
- Cyclomatic complexity. This is too many branches (if statements) or loops within loops. Complex reasoning is difficult to understand and should be broken up.
- Too many parameters. If there are lots of parameters for a method it is a sign that they might work better as an object. This is especially true if those same variables are seen together more than once. If variables travel together, they are coupled and should be encapsulated as an object.
- Magic Numbers. This term refers to literals of any kind in the code. These values are good candidates for constants. They generally are used multiple times in the code and are likely to be modified in future versions of the code.

---

[2] While "is-a" and "has-a" are well known terms you will find in textbooks, "acts-as-a" is not. That's a Shannon original.

[3] A good answer includes more technical details, like the fact that objects can implement multiple interfaces but not multiple superclasses, and superclasses have state and interfaces usually do not.

- Too few or too many comments. Too few comments means that code is hard to understand, and too many comments sometimes means that code is too complex or the variable and method names aren't self-explanatory. I rarely see students use too many comments, but I often see too few. Do NOT wait until your code is finished before you comment.

In addition to these well-known code smells, here are some I find particularly annoying:

- Long if statement chain (if … else if… else if… else if… or a switch statement). This is one form of "cyclomatic complexity" as stated above. It almost always has some duplication in it, and it is open to modification. The fix for long if statements is almost always to change the design to use a map or an inheritance structure.
- Using misleading code, like a number that is either 0 or 1 rather than a boolean. I have seen students use String constants "1", "2", etc… This is unnecessarily confusing.
- ```
  if(foo){
       return true;
  }
  else{
       return false;
  }
  ```

  I see this code all too often. It is unnecessarily complex and shows a lack of understanding of how to use booleans. This logic should be written simply as:

  ```
  return foo;
  ```

- Public instance variables and methods. If no one else needs it, make it private. Protect your code from inappropriate intimacy.
- Unnecessary getters and setters. Do not get in the habit of automatically making getters and setters for all your instance variables. If you do that, then you may as well have made them public. Write them only as needed. Resist the urge to provide vanilla setters. At the very least, make sure the given value is appropriate.
- Instance variables that should be local. An instance variable is supposed to be an attribute variable. That means it should be a vital attribute to the class as a whole. Students mistakenly think all large data structures should be instance variables, but this is only true if it prevents duplicated code or makes sense intuitively as an instance variable. It is preferable to have local variables and to pass values through parameters and returns of method calls. Remember that making variables instance variables opens up their scope so they can be modified by other pieces of code.
- Use of the keyword "instanceof". Using this keyword almost always shows a place where an inheritance hierarchy should be used.
- Overuse of the "static" keyword. There is nothing inherently wrong with static variables or methods if used correctly. However, static variables are sometimes used like global variables. I also often see students make something static and when I ask them why the answer is "eclipse told me to" or "I have to for this to compile." These aren't good reasons – they are indications of a lack of understanding.
- Long main methods or classes. The main method is, by definition, static! It generally should be just the starting point of the application and not include logic (that cannot be reused). It should be in its own class (called Main) so that many main methods can be implemented and you can run a portion of the application for testing.
  ```

When I grade your code, I will have a list of code smells that I'm looking for. Finding them shows me the design flaws. You should do this with your own code. Once you find the flaw, try to fix it. If you can't, ask me and I will be happy to help you.

A Caveat

Sometimes we tend to be all-or-nothing thinkers. You really can't approach code smells (or life) this way. There are times when it is legitimate to make a long method or public instance variables or even switch statements[4]. Sometimes design is a tradeoff between competing principles and you have to choose the lesser of evils. Code smells are simply indications that problems *might* exist in the design. If I see them, you will have to justify them to me. I find that 90% of the time, the code smell is a better design trying to get my attention.

Final Advice

You have now learned the vocabulary of code design. Did you notice how forceful the vocabulary is? (STUPID, SOLID, smelly?) This stuff matters, and people feel very strongly about it. Now that you know the principles, you need to practice refactoring. This document gives facts but few examples. You can find plenty of them online,[5] but you really learn refactoring by doing it. At first it will seem like a lot of work to refactor your code before you turn it in, but you will get faster with practice.

I would like you to cultivate the habit of writing code in small pieces. You should write, test, debug, and comment each piece before you move on to the next. Doing so helps you think about the design as you build and saves you time debugging in the long run. Resist the urge to start at the last minute, panic, and bang out a bunch of lousy code at the last minute. Better quality of life* emotional high, rhythm of working, lack of general confusion, lack of high stress/high panic situations

Remember that we are aiming for a higher standard than "functional" code. Over time you will learn to adhere to better design faster and your code will be much better for it.

---

[4] Don't tell Joel I said that.

[5] Google "Java refactoring examples"