Lecture Notes
November 15, 2018

A design principle is a general technique applied to writing code to avoid bad design

Bad design is:

- Rigid - Hard to change because every change affects too many other parts of the system.
- Fragile - When you make a change, unexpected parts of the system break.
- Immobile - Hard to reuse in another application because it cannot be disentangled from the current application.

You Aren't Gonna Need It (YAGNI): do the simplest design that could possibly work

Single Responsibility Principle (SRP): Every object or class should have a single responsibility, and all its tasks or services should be focused on that responsibility (same idea as cohesion)

Seemingly contradicts the multiple "responsibilities" of CRC cards, but there refers to tasks or services that are all part of the same focused single responsibility

Only one reason to change: When the requirements of the project change, the modifications will be visible through the alteration of the responsibilities of one or more classes. If a class has several responsibilities, then it will have more reasons to change

Solution: Break up the functionality into multiple classes, where each individual class does only one thing - leads to more smaller classes

How to tell when you're violating SRP: Does it make sense to say "The <class name> <method name> itself" for each method? (after fixing grammar)

Example

Don't Repeat Yourself principle (DRY): Avoid duplicate code and redundant work, instead abstract out common code and put in a single location. Make sure that you only implement each feature and requirement once

Do not copy/paste - if you find code that you would like to copy/paste (from your own codebase), remove from its original location and put in a new method, then both places should call that method

When similar code was written independently, detect and combine during refactoring

Some IDEs and refactoring tools automate "code clone" detection, but this only works if the code *looks* similar, not if it *behaves* similarly but looks different - but similarly behaving code requires redundant work whether or not it looks similar

DRY is about putting a piece of functionality in one place vs. SRP is about making sure a class does only one thing

Example

Open-closed principle (OCP): Classes, modules and functions should be "open for extension and closed for modification", allow change but without modifying any existing code

Once you have code that works, and is being used, you don't want to make changes to it unless you have to

Why not? Backward compatibility, regression testing

But since requirements change, we need to be able to change behavior

- Change the behavior of an existing class by deriving a new subclass
- Change the behavior of an existing function by adding a wrapper around that function
- Change the behavior of a library by adding new API functions or a new implementation of the same interface

Example

Liskov Substitution Principle (LSP): Extends OCP to require that subtypes must be substitutable for their base types (polymorphism)

Do inherited methods make sense for instances of the derived class? Do overridden methods take the same parameter types and return the same types as the base class?

If not, then we are not actually inheriting - instead should delegate any common functionality, and if there is no common functionality then there's no relationship at all so why pretend?

Code already using references to the base classes should be able to use instances of the derived classes without being aware of the switch

"Design by Contract" - The contract of a method informs the author of a class about the behaviors that she can safely rely on. The contract could be specified by declaring preconditions and postconditions for each method. The preconditions must be true in order for the method to execute. On completion, after executing the method, the method guarantees that the postconditions are true. Derived classes need to fulfill the same contract.

Example

There are other “design principles” in the literature

These seem (to me) the most generally applicable to agile processes, with no grand design up front

Read for Tuesday:
software architectural patterns
and
textbook, chapter 12 Test Implementation

Next (and last) pair assignment Bug Hunt 2 due Tuesday, November 20, 10:10am

Second Iteration due Tuesday November 27, 11:59pm

Second Iteration Demo due Tuesday December 4, 11:59pm

Second Individual Assessment will become available on December 4

Advertisement for my Spring 2019 course,
6156 Topics in Software Engineering

6156 is not "more" 4156, and not "more advanced" 4156.

4156 is about doing software engineering, and 6156 is about studying and improving software engineering.

6156 is essentially a research seminar, although most students who take it do not have any plans for PhD. Instead they aspire to be "technology leaders" in industry, introducing the latest greatest new tech to their team.

As a "Topics In" course, 6156 can the same semester have multiple sections with different topics taught by different people. For example, Prof. Ferguson teaches some specific latest greatest new tech when he teaches a 6156 section.

In the past, when I've taught it, I've let students choose their own topics (within constraints, e.g., I have to consider it "software engineering"). I have not yet decided for spring, this time I might focus on some specific topic. Taking suggestions ....

Sample 6156 midterm papers and final projects available here

Also seeking a TA who will take the course concurrently (or has already taken).