Object-Oriented Analysis And Design

# Win your customers over,

Customers will think your software is great when it does what it’s supposed to do.

# Make your code as smart as you are,

You (and your co-workers) will think your software is great when it’s easy to maintain, reuse, and extend.

# Great software in 3 easy steps

1. Make sure your software does what the customers want it to do.

2. Apply basic OO principles to add flexibility.

3. Strive for maintainable, reusable design.

# Don’t create problems to solve problems

# Use the textual description of the problem you’re trying to solve to make sure that your design lines up with the intended functionality of your application.

# Anytime you see duplicate code, look for a place to encapsulate!

# Delegation,

The act of one object forwarding an operation to another object, to be performed on behalf of the first object.

# Flexibility,

Use me so that your software can change and grow without constant rework. I keep your application from being fragile.

# Encapsulation,

You use me to keep the parts of your code that stay the same separation from the parts that change then it's really easy to make changes to your code without breaking everything.

# Functionality,

Without me, you’ll never actually make the customers happy. No matter well-designed your application is, I’m the thing that puts a smile on the customer’s face.

# Design Pattern,

I’m all about reuse and making sure you’re not trying to solve a problem that someone else already figured out.

# So what exactly is a requirement, anyway?

It’s a specific thing your system has to do to work correctly.

# Requirement,

A requirement is a singular need detailing what a particular product or service should be or do. It is most commonly used in a formal sense in system engineering or software engineering.

# The best way to get good requirements is to understand what a system is supposed to do.

# A use case describes what your system does to accomplish a particular customer goal.

# Use case,

A use case is a technique for capturing the potential requirements of a new system or software change. Each use case provides one or more scenarios that convey how the system should interact with the end-user or another system to achieve a specific goal.

# One use case, three parts

1. Clear value, every use case must have a clear value to the system. If the use case doesn’t help the customer achieve their goal, then the use case isn’t of much use.
2. Start and stop, every use case must have a definite starting and stopping point. Something must begin the process, and then must be a condition that indicates that the process is complete.
3. Every use case is started off by an external initiator, outside of the system. Sometimes that initiator is a person, but it could be anything outside of the system.

# Your system must work in the real world… so plan and test for when things go wrong.

# External initiator,

Kicks off the list of steps described in a use case. Without this, a use case never gets going.

# Use case,

Helps you gather good requirements. Tells a story about how a system works.

# Start condition,

This is always the first step in the use case.

# Requirement,

Something a system has to do to be a success.

# Clear value,

Without this, a use case isn’t worth anything to anyone. Use cases without this always fail.

# Stop condition,

Lets you know when a use case is finished. Without this, use cases can go on forever.

# Main path,

How a system works when everything is going right. This is usually what customers describe when they’re talking about the system.

# Requirements always change if you’ve got good use cases, though, you can usually change your software quickly to adjust to those new requirements.

# A complete path through a use case, from the first step to the last, is called a scenario. Most use cases have several different scenarios, but they always share the same user goal.

# Sometimes a change in requirements reveals problems with your system that you didn’t even know were there.

# Change is constant, and your system should always improve every time you work on it.

# Analysis helps you ensure your system works in a real-world context.

# Delegation shields your objects from implementation changes to other objects in your software.

# Looking at the nouns (and verbs) in your use case to figure out classes and methods is called textual analysis.

# A good use case clearly and accurately explains what a system does, in language that’s easily understood. With a good use case complete, textual analysis is a quick and easy way to figure out the classes in your system.

# Abstract classes are placeholders for actual implementation classes. The abstract class defines behavior, and the subclasses implement that behavior.

# Whenever you find common behavior in two or more places, look to abstract that behavior into a class, and then reuse that behavior in the common classes.

# Coding to an interface, rather than to implementation, makes your software easier to extend.

# By coding to an interface, your code will work with all of the interface’s subclasses-even ones that haven’t been created yet.

# Each class in your application should have only one reason to change.

# When you have a set of properties that vary across your objects, use a collection, like a Map, to store those properties dynamically. You’ll remove lots of methods from your classes, and avoid having to change your code when new properties are added to your app.

# A cohesive class does one thing really well and does not try to do or be something else.

# Cohesion,

Cohesion measures the degree of connectivity among the elements of a single module, class, or object. The higher the cohesion of your software is, the more well-defined and related the responsibilities of each individual class in your application. Each class has a very specific set of closely related actions it performs.

# Commonality,

What is the system like? One way you can find out more about a system is to figure out what the system is LIKE. In other words, are there some things that you do know about that system functions or behaves like?

# Variability,

What is the system not like? Another great way to find out what a system should do is to figure out what it’s NOT like. This helps you determine what you DON’T NEED to worry about in your system.

# But what is a feature, anyway?

A feature is just a high-level description of something a system needs to do. You usually get features from talking to your customers.

# Get features from the customer, and then figure out the requirements you need to implement those features.

# Always defer details as long as you can.

# Domain analysis lets you check your designs, and still speak the customer’s language.

# Domain Analysis,

The process of identifying, collecting, organizing, and representing the relevant information of a domain, based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain.

# Architecture is your design structure and highlights the most important parts of your app and the relationships between those parts.

# Architecture,

Architecture is the organizational structure of a system, including its decomposition into parts, their connectivity, interaction mechanisms, and the guiding principles and decision that you use in the design of a system.

# Architecture takes a big chaotic mess… and helps us turn it into a well-ordered application.

# The things in your application that are really important are architecturally significant, and you should focus on them first.

# The three Qs of architecture,

When you’re trying to figure out if something is architecturally significant, there are three questions you can ask:

1. Is it part of the essence of the system?
2. What the fuck does it mean?
3. How the fuck do I do it.

# The essence of a system is what that system is at its most basic level.

# Using the three Qs of architecture, we’ve started to add some order to all that confusion we started out with,

1. Remember all this? Quite a mess when we started out…
2. But then we focussed on making the system do what it is supposed to do.
3. Finally, we’ve narrowed that down to just a few key features to focus on.

# The point here is to REDUCE RISK, not to argue over which key feature you start with first. You can start with any of these, as long as you’re focused on building what you’re supposed to be building.

# Focus on one feature at a time to reduce risk in your project. Don’t get distracted with features that won’t help reduce risk.

# Sometimes the best way to write great code is to hold off on writing code as long as you can.

# Design principle,

A design principle is a basic tool or technique that can be applied to designing or writing code to make that code more maintainable, flexible, or extensible.

# Open-closed Principle,

Classes should be open for extension and closed for modification.

# Don’t repeat yourself,

Avoid duplicate code by abstracting out things that are common and placing those things in a single location. DRY is about having each piece of information and behavior in your system in a single, sensible place.

# Single Responsibility Principle,

Every object in your system should have a single responsibility, and all the object’s services should be focused on carrying out that single responsibility.

# Liskov Substitution Principle,

Subtypes must be substitutable for their base types. The LSP is all about well-designed inheritance. When you inherit from a base class, you must be able to substitute your subclass for that base class without things going terribly wrong. Otherwise, you’ve used inheritance incorrectly.

# Delegation,

Is when you hand over the responsibility for a particular task to another class or method.

# Use composition to assemble behaviors from other classes.

# Composition,

Allows you to use behavior from a family of other classes, and to change that behavior at runtime.

# Aggregation,

Is when one class is used as part of another class, but still exists outside of that other class.

# If you favor delegation, composition, and aggregation over inheritance, your software will usually be more flexible, and easier to maintain, extend, and reuse.

# You write great software iteratively. Work on the big picture, and then iterate over pieces of the app until it’s complete.

# Feature-driven development,

…is when you pick a specific feature in your app, and plan, analyze, and develop that feature to completion.

# Use case-driven development,

…is when you pick a scenario through a use case and write code to support that complete scenario through the use case.

# You should test your software for every possible usage you can think of. Be creative! Don’t forget to test for incorrect usage of the software, too. You’ll catch errors early, and make your customer very happy.

# Programming by contract,

When you program by contract, you and your software’s users are agreeing that your software will behave in a certain way.

# Defensive programming,

Defensive programming assumes the worst and tries to protect itself (and you) against misuse or bad data.

# When you are programming by contract, you’re working with client code to agree on how you’ll handle problem situations.

# When you’re programming defensively, you’re making sure the client gets a “safe” response, no matter what the client wants to have to happen.

# Your feature lists are all about understanding what your software is supposed to do.

# Your use case diagrams let you start thinking about how your software will be used, without getting into a bunch of unnecessary details.

# Use case connects what your app does to how it will be used.

# You need to make sure that your use cases match up with the features you have to deliver to the customer.

# Use cases reflect usage, features reflect functionality.

# The features in your system reflect your system’s functionality. Your system must do those things in order for the use cases to actually work, even though the functionality isn’t always an explicit part of any particular use case.

# You should only expose clients of your code to the classes that NEED to interact with.

# Classes that the clients don’t interact with can be changed with minimal client code being affected.