# Object Oriented Design & Analysis - Planning Template

This document serves as a template for documentation of the planning process for a project using object oriented design and analysis.

# The Project Lifecycle

The following is an outline of the lifecycle of a project or one of its components:

## Single Iteration (Typically)

1. **Feature List** (Statisfy Customer)
   1. Figure out what your app is supposed to do at a high level.
2. **Use Cases** (Statisfy Customer)
   1. Nail down the big processes that your app performs, and any external forces that are involved.
3. **Break Up the Problem** (Statisfy Customer)
   1. Break your application up into modules of functionality, and then decide on an order in which to tackle each of your modules.

## Iterative Development

1. **Requirements** (Statisfy Customer)
   1. Figure out the individual requirements for each module, and make sure those fit in with the big picture.
2. **Domain Analysis** (Statisfy Customer)
   1. Figure out how your use cases map to objects in your app, and make sure your customer is on the same page as you are.
3. **Preliminary Design** (OOP)
   1. Fill in details about your objects, define relationships between the objects, and apply principles and patterns.
4. **Primary Implementation** (OOP)
   1. Write code, test it, and make sure it works. Do this for each behavior, each feature, each use case, each problem, until you're done.
5. **Secondary Implementation** (Strive for maintainable & reusable design)
6. **Delivery**

# Feature List

A feature is a high level description of what a system needs to do. Usually one feature will result in multiple requirements.

## Talk to the customer

Talk to the customer to determine what are the things that the program should do.

## Features List

Describe in plain English precisely what the system should do.

1. Feature 1
2. Feature 2

# Use Cases

A use case describes what the system does to accomplish a particular customer goal using non-programming terminology. It 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.

Every use case must have:

1. **Clear value** to the system
2. A definite **starting** and **stopping** point. Something must begin the process (external initiator), and then there must be a condition that indicates that a process is complete.
3. An **external initiator**, outside of the system, to start off the use case. Sometimes the initiator is a person, but it could be anything outside of the system.
4. [Optional] **Alternate Paths** for any other way that the use case can go when things don't go according to plan.

One of the aims of use cases is to develop **Requirements Lists**.

### Alternate Paths vs. Happy Path

Primary paths (Happy Paths) are the main path for a given use case. Alternate paths are the paths taken when unexpected issues occur along a given use case.

## Talk to the customer

Talk to the customer in order to work out as many use cases as possible. Carefully vet the cases to boil them down to unique cases that accurately list all of the necessary steps and needs.

## Sample Use Case Sheet

**Primary Actor:** (External Initiator)

**Secondary Actors:** Possibly interact with system

**Pre-Condition:** Starting context of the use case

**Goal:** State the aim of the use case that delivers value

**Main Path:**

1. Start Condition
2. ...

2.1. SubStep of 2

1. End Condition

**Alternative Paths:**

2.a. Alternate path to 2.

2.1.a. Alternate path to 2.1.

2.1.b. Alternate path to 2.1.

2.2.b. Alternate path to 2.2 within alternate path b.

## Sample Use Case Sheet (Alternate)

|  |  |  |
| --- | --- | --- |
| **Main Path:** | **Alternate Paths:** |  |
| 1. Start Condition |  |  |
| 2. | 2.a. |  |
| 2.1 | 2.1.a | 2.1.b |
| 2.2 |  | 2.2.b |
| 3. End Condition |  |  |

## Use Case Diagram

Use case diagrams can be helpful to see what a system does without getting into all of the detail that use cases require. Usually the diagrams include the following:

1. **Box** representing the **system**
2. **External actors**, drawn outside the box
3. **Use case titles** as bubbles within the system box.
4. **Lines** establishing which actors are related to which use cases.

Use case diagrams help to keep an eye on the fundamental things that the system must do and avoid getting caught up in the details. They should account for all of the features in a system.

# Break Up the Problem

## Solving Big Problems

1. Listen to the customer and figure out what they want you to build.
2. Put together a feature list, in language the customer understands.
3. Make sure your features are what the customer actually wants.
4. Create blueprints of the system using use case diagrams (and use cases).
5. Break the big system up into lots of smaller sections (e.g. Features -> Requirements)
6. Apply design patterns to the smaller sections of the system.
7. Use basic OOA&D principles to design and code each smaller section.

## Architecture

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

Is the app design structure and highlights the most important parts of the app and the relationships between those parts.

1. Make sure your software does what the customer wants it to do.
   1. Get Features List
   2. Determine Key Features from the list and work on them first (MVP).
   3. Ideally, identify the order to work on the key features by determining their relative risk to the success of the project.
   4. Focus on one feature at a time to reduce risk in the project.
2. Apply basic OO Principles to add flexibility.
3. Strive for maintainable, reusable design.

## Key Features List

Features that are architecturally significant.

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

1. Is it part of the essence of the system? Can you imagine the system without the feature?
2. What does it mean?
   1. Anytime you're unsure about what something is, it could take lots of time, or create problems with the rest of the system. Send time on these features early, rather than late.
   2. Steps to address:
      1. Ask the customer (What does the feature mean?)
      2. Commonality analysis (How do I realize that feature in my system?)
      3. Implementation Plan
3. How do I do it?
   1. Focus early attention on features that seem really hard to implement, or are totally new programming tasks.

## Analysis & Design

* Well-designed software is easy to change and extend.
* Use basic OO principles to make your software more flexible.
* If a design isn't flexible, then **CHANGE IT!** Never settle on bad design.
* Make sure each of your classes is cohesive: Each class should focus on doing **ONE THING** really well.
* Always strive for higher cohesion as you move through your software's design life cycle.

## Analysis

### Domain Analysis

Represents a system in language that the customer will understand, such as:

1. Features List
2. Key Features List
3. Use Cases
4. Use Case Diagrams
5. Scenarios

### Textual Analysis

Analyze the text of the Key Features List and/or the Use Case to figure out classes and methods. Typically **nouns** correspond to **classes** and **verbs** correspond to **methods**.

Even if a noun does not correspond to a class, pay close attention to them. They are what you should focus on.

### Commonality Analysis

Determine commonality in the problem. This can indicate what are the main classes to develop, and the more abstract ways to define aspects of the program and its behavior. It also helps to better lump together various requirements and satisfy DRY in requirements and coding.

### Variability Analysis

Determines what is unique in the system that requires unique individual attention.

On the class level it helps determine inheritance trees & needs, as well as the composition and characteristics of class **properties**.

### SRP Analysis

Do this to help apply the SRP. For any lines that do not make sense, it indicates that the method belongs in another class.

SRP Analysis for [class name]

1. The [class name] [method][s] itself.
2. The [class name] [method][s] itself.
3. Repeat for all methods

# Requirements

Requirements are lower-level than requirements and describe specifically what needs to be developed in order to accomplish the desired features. They can be drawn from the Key Features List and/or scenarios from Use Cases.

* Good requirements ensure your system works like your customers expect.
* Make sure your requirements cover all of the steps in the use cases for your system.
* Use your use cases to find out about things your customers forgot to tell you.
* Your use cases will reveal any incomplete or missing requirements that you might have to add to your system.
* Your requirements will always change (and grow) over time.

## Scenarios

Each requirement usually satisfies a particular scenario. A scenario would be a particular path chosen through a single Use Case. A Use Case might have multiple scenarios. For example, based on the use cases described earlier, scenarios could be mapped out as:

1. 1, 2, 2.1, 2.2 .... , 3
2. 1, 2, 3
3. ...

## Requirements List

For each step of the scenario, list what the requirements are and note whether or not they are to be addressed by the program vs. external actors.

|  |  |
| --- | --- |
| **Scenario 1** | **Requirement** |
| 1 |  |
| 2 |  |
| 2.1 |  |
| 2.2 |  |
| 3. |  |

|  |  |
| --- | --- |
| **Scenario 22** | **Requirement** |
| 1 |  |
| 2 |  |
| 3. |  |

# Preliminary Design

## Design Pattern

Generic style of programming that is typically used to solve common programming problems. Don't follow patterns blindly, but use them when appropriate as guides to programming strategies.

## Design Principles

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

### Object-Oriented Principles

1. Encapsulate what varies.
2. Code to an interface rather than to an implementation
3. Each class should have only one reason to change.
4. Classes are about behavior and functionality.
5. **OCP (Open-Closed Principle)**
   1. Classes should be open for extension, and closed for modification.
   2. It is a combination of encapsulation and abstraction.
   3. Done through:
      1. Inheritance
      2. Adding new public methods to use private methods in new ways.
      3. Aggregation
6. **DRY (Don't Repeat Yourself)**
   1. Avoid duplicate code by abstracting out things that are common and placing those things in a single location.
   2. Not just about avoiding duplicate code, but also avoiding duplicate implementation of features and requirements.
   3. DRY is about having each piece of information and behavior of a system in a single, sensible place.
   4. Apply this not just to programming, but also the use cases and scenarios before programming begins in order to have the most basic, unique cases outlined. Redundancy can hide in similar but differently-worded use cases.
7. **SRP (Single Responsibility Principle)**
   1. Every object in the system should have a single responsibility, and all of the object's services should be focused on carrying out that single responsibility.
   2. Implemented correctly when each class has only one reason to change.
   3. Creates highly **cohesive** software.
8. **LSP (Liskov Substitution Principle)**
   1. Subtypes must be substitutable for their base types.
   2. All about well-designed inheritance.
   3. When inheriting from a base class, you must be able to substitute your subclass for that base class without things going wrong.
   4. If inheritance violates this, desired behavior can often be fixed by:

### Alternatives To Inheritance

If inheritance violates an OO Principle (especially LSP), the following are great ways to achieve programming aims rather than improper inheritance:

1. **Delegation**
   1. When you hand over the responsibility for a particular task to another class or method.
   2. Best used when you want to use another class' functionality, as is, without changing that behavior at all.
2. **Composition**
   1. Assembles behaviors from other classes.
   2. Allows you to use behavior from a family of other classes, and to change that behavior at run time.
   3. For example, implementing one or more interfaces.
   4. The object **owns** the behaviors, and when the object is destroyed, so are all of its behaviors, since they are manifest as methods of the object.
   5. Best to use when you want to use behavior defined in an interface, and then choose from a variety of implementations of that interface, both at compile time and run time.
3. **Aggregation**
   1. When one class is used as part of another class, but still exists outside of that class.
   2. Best to use when you want to use the behavior of another object, but that object is to exist outside the existence of the calling object.

### 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.

### Coupling

The degree to which changes in one object requires other changes in other objects.

### Delegation

The act of one object forwarding an operation to another object, to be performed on behalf of the first object. This helps keep an application loosely coupled since changes to one object doesn't requires changes elsewhere (as long as the object's interface is left intact).

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

## Class Diagram

Typically a program is designed through UML class diagrams before coding begins. This helps to translate requirements and use cases to programming, and provides an intermediate means of communicating to the client what the programming implementation was.

|  |  |
| --- | --- |
| **UML Term** | **Programming Term** |
| Abstract Class | Abstract Class |
| Association | Relationship |
| Generalization | Inheritance |
| Aggregation | Aggregation |
| Attribute | Variable |
| Operation | Method |
| Generalization | Inheritance |

# Implementation

## Iteration Approaches

### Test Driven Development

Focuses on getting the behavior of classes right by writing tests first, and then programming classes to pass the tests. Usually done as a means of implementing either Feature or Use Case Driven Development.

### Feature Driven Development

When you pick a specific feature in your app, and plan, analyze and develop that feature to completion. This is a more granular approach.

* Works well when you have a lot of different features that don't interconnect a lot.
* Allows you to show the customer working code faster. A better choice to use for a customer who is impatient to see results.
* Is very functionally-driven. You're not going to forget about any features using this method.
* Works particularly well on systems with lots of disconnected pieces of functionality.

### Use Case Driven Development

When you pick a scenario through a use case, and write code to support that complete scenario through the use case. This is a more "big picture" approach.

* Works well when your app has lots of processes and scenarios rather than individual pieces of functionality.
* Allows you to show the customer bigger pieces of functionality at each stage of development.
* Is very user-centric. You'll code for all the different ways a user can use your system with use case driven development.
* Works particularly well on transactional systems, where the system is largely defined by lengthy, complicated processes.

## Test Scenarios

A testing scenario may result in more than one testing case. Each test case should have:

1. An ID and a name.
2. One specific thing that it tests.
3. An input that you supply.
4. An output that you expect.
5. A starting state.

## Test Case Design

Below is a sample diagram of a good way to briefly design and describe tests. This can be good to fill in before coding tests.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Name (What We're Testing) | Input | Expected Output | Starting State |
| 1 | Setting/Getting the type property | "Type",  "infantry" | "type",  "infantry" | Existing Unit object. |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |

# Delivery