Introduction to Software Design

This article introduces the concept of software design as well as some of its important decisions.

Have you ever wondered how your code was supposed to fit in with the rest of the application? Well, then it's good that you're here because we'll explain how software design plans how application code works together.

Software design can involve the creation of flowcharts, *Unified Modeling Language* (UML) diagrams, interface descriptions, design patterns, as well as other practices and tools. All of these tools help us create an organized approach to creating applications that are clearly defined, flexible, and maintainable.

When we talk about the design of all of our components, our *system*, we're discussing *software architecture*. We want to model our system and be able to break up that model into pieces.

When we design for software architecture, we ask questions like:

- How will we organize our software?
- How will the parts talk with each other?
- How will we move and store the data?

These types of questions think about how the pieces of our app work at a high level — which will help guide our team along as we make changes to our code.

Designing For Change

It is easy at first to make changes without considering the design. However, each addition makes our system more complex. Over time, it becomes harder to add new pieces. When this happens, teams either have to redo parts of the code or sometimes start over completely.

We can use design guidelines/tools to avoid this problem and make our system easier to change over time. Some factors that influence a system's ability to change include:

- Each system part should have a clear purpose.

- The things most likely to change should be separate from the rest of the system.
- Classes should have a clear way of interacting with them.
- Our system should use standard design patterns and practices.

Multiple Choice Question
As we develop our application, we are not sure which type of database we are going to use. How might we design around this problem?
Keeping the database code interactions separate from the rest of the system.
Designing a huge variety of ways to interact with the database.
Having many different system components interact with the database system.
Picking a database and sticking with it regardless of changes.
👏
Correct! We want very little of our system to interact with the database, which is likely to change, so that replacing it is simpler.

Some General Rules

Some popular adages can be important to keep in mind as you design systems.

YAGNI: You Aren't Going to Need It

The *YAGNI* principle, "You Aren't Going to Need It" implores us to design our system for only the features that we currently are going to work on. Oftentimes we are tempted to add in pieces to prepare for planned features.

Imagine we currently are making a single-player game, but we decide to add an unused networking system just in case we decide to add multiplayer in the future. YAGNI argues against this, that the networking functionality should be added only when necessary. YAGNI aims to keep the number of system pieces low and reduce the chance of developing pieces that we are never going to use.

KISS: Keep It Simple, Silly

KISS, or "Keep it Simple, Silly" directs us to keep the designs of our systems as simple as they possibly can be. Sometimes we are tempted to overly flex our design muscles, creating systems that are extremely complex or elaborate. However, complexity is not the desired goal of a system, as complexity makes

a system harder to understand and change. Instead, we should seek to have our system as simple as possible, only adding complexity when necessary.

DRY: Don’t Repeat Yourself

The *DRY* Principle can be rephrased as “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system”. Essentially this is saying that we should look for places in our system where we are representing the same information or knowledge multiple times. Whenever these places are found, we should seek to replace them, having the information defined in a single place and having the original pieces depend on that new single source.

While it is important to consider DRY for areas of an application in which repetition is occurring, overly applying these principles can introduce needless additional work and complexity. Optimization and abstraction should be done in areas in which it is useful, rather than in every place possible.

Multiple Choice Question
We are designing a system and are considering adding in the beginnings of a feature that we might implement next year. Which rule of thumb is most applicable to this situation?
UML
YAGNI
DRY
OOP

Correct! YAGNI, or You Aren’t Going to Need It, asks us to only design for features that are immediately going to be implemented. Features such as this should not be added to our system until necessary. Show less

Review

In this article, we introduced software design, architecture, and some of their key concepts. Over the next several articles we will dive deeper into different aspects of software design.