**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.

**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. 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.

**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?**

* 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

**Introduction to OOP and UML**

**This article introduces aspects of OOP relevant to system design, with a focus on encapsulation, abstraction, polymorphism, and inheritance.**

We live in a world of objects and behaviors. A dog barks at a cat. A passenger drives a car. When working with software that relates to our world, it makes sense to define our code in terms of the objects and behaviors we are working with.

The introduction of object-oriented programming, or OOP, was a major shift in software design, allowing us to define our software components as objects. Before OOP, the main language type was imperative programming, in which blocks, functions, and files were the main methods of component organization. OOP added new levels of structure on top of imperative programming and introduced four pillars to assist in creating reusable and extensible designs: abstraction, encapsulation, inheritance, and polymorphism.

In this article, we will explain how these pillars influence system design and describe using Unified Modeling Language, or UML, to represent objects and their relationships.

**UML: Describing an Object and Its Relationships**

To describe how an object is going to work in our system, we should not need to see every line of code. The names of classes, methods, and attributes should be able to convey the purpose of an object. Working with our objects in high-level diagrams allows us to reason about how our components will work together without having implemented them, or knowing all of the technical details.

The UML base class diagram was designed to document objects at the name, method, and attribute level. Below is an example of a UML class diagram. At the top is the class name, followed by the attributes, and then the methods:

These diagrams are meant to convey what an object is for, as well as how it is going to be interacted with. The diagrams can also give a developer some idea of how it should be implemented.

**Which is NOT included in a class UML diagram?**

* Implementation details. UML diagrams represent a class at a high-level, leaving the implementation details up to the code.

**System Diagrams and Flowcharts**

We can also represent objects at a system level, displaying the relationships between objects. Take a look at the following example of a system diagram:

This diagram describes how information flows between the main pieces of the application at a high level. The Website receives user requests, which it forwards to a Server. The Server processes those requests, turning them into Database queries. The Database sends information back to the Server, which packages content to return to the Website. These kinds of diagrams can help gain a bird’s-eye view of how a complicated system works.

**Designing OOP Classes**

These diagrams give us a nice way to represent our classes, but how do we ensure these classes are well designed?

Let’s start from the main pillars of OOP:

* Abstraction
* Encapsulation
* Inheritance
* Polymorphism

**Abstraction**

**Abstraction** is hiding the low-level details and representing an idea at a high level. When we design an object’s methods that will be used by other objects, we want to abstract what the methods will do at the highest level possible. This allows other classes to only depend on the “big picture” of a class, which is less likely to change over time.

**Encapsulation**

**Encapsulation** is highly related to abstraction. Encapsulation means that an object should manage its internal state, exposing only behaviors to be used by other objects. Essentially, only the high-level abstract behaviors of an object should be accessible to other objects, and access to an object’s internals should be restricted.

**Inheritance**

**Inheritance** means that an object inherits characteristics of its parent object. Inheritance allows us to implement behaviors shared by subclasses in the parent object, and leave the implementation of behavioral differences to a child object.

**Polymorphism**

By defining classes using inheritance, we can reuse the aspects of objects that stay the same. With our next pillar, **polymorphism**, we can define the differences between subclasses. Polymorphism allows a class to overwrite the inherited behavior of its superclass. **Polymorphism** allows the behavior of particular superclass methods to be left up to the child class implementation.

**Which pillar of Object-Oriented Programming would be described as hiding an object’s internal implementation details behind higher-level methods?**

* Encapsulation

**Review**

In this article, we discussed the role that OOP plays in system design as well as the basics of representing OOP systems in UML and system diagrams. We further elaborated on the meaning and purpose of abstraction, encapsulation, inheritance, and polymorphism.

* **Video resource** [**here**](https://www.youtube.com/watch?v=u8gRq4OojXY)

**The SOLID Principles**

**This article introduces the SOLID principles of OOP design.**

Need some guidelines for creating well-designed classes? In this article, we will introduce some principles that provide just the solution. The SOLID Principles aim to create software that is easy to change, understand, and reuse.

SOLID is an acronym that stands for:

* **Single Responsibility Principle**: Classes should have only one reason to change.
* **Open-Closed Principle**: We should be able to add functionality through the creation of new classes rather than changing existing classes.
* **Liskov Substitution Principle**: We should be able to replace a subclass with its parent class without breaking our codebase.
* **Interface Segregation Principle**: Interfaces should only contain attributes and methods used by all of their subclasses.
* **Dependency Inversion Principle**: Classes should interact with interfaces rather than concrete classes.

**Single Responsibility Principle**

The Single Responsibility Principle states that **modules (a class or a set of functions) should have only one reason to change**. Reasons to change include business decisions, database changes, or adjustments to the user experience. When multiple of these factors could impact a class, that class is violating the Single Responsibility Principle. In order to align with the Single Responsibility Principle, the class would be broken up into smaller classes according to these different reasons. This reduces the complexity of our classes and the number of times each will have to change.

Let’s say we are creating a tower defense game, and we need to design an Enemy entity. We initially come up with an object that handles the following:

* Attacking and taking damage
* Displaying on the screen
* State info such as health and type information

At one level, it seems our class follows the Single Responsibility Principle, if something about the Enemy changes, then this class has to change. However, each feature corresponds to completely different areas of the application. Changes to the gameplay mechanics, presentation, or database can each result in changes to this class.

The Single Responsibility principle would have us split the class into something like the following:

* **EnemyBehavior**: Handles attacking and taking damage.
* **EnemyState**: Handles the state information for the enemy.
* **EnemyStorage**: Handles its database storage and retrieval.
* **EnemyDisplay**: Handles displaying the enemy on the screen.

Overall, the Single Responsibility Principle minimizes the impact of change between files.

**Open-Closed Principle**

The Open-Closed Principle states that **software classes should be open for extension but closed for modification**. This means we should be able to add functionality to our system by adding new classes without having to change existing ones.

When we have to edit an existing class in order to incorporate new business features, our code likely violates both the Single Responsibility and Open-Closed Principles. The combination of the Single Responsibility and Open-Closed Principles has us focus on separating our classes based on independent behaviors.

**Which would be a side effect of utilizing the Open-Closed and Single Responsibility Principles?**

* A system consisting of a large number of small classes. Applying these principles will increase the number of classes in your system, as each class will have limited responsibilities and extensions will be frequently created.

**Liskov Substitution Principle**

The Liskov Substitution Principle states that **a subclass should be able to be substituted with its parent class**. There should be no special behavior that we have to do when dealing with particular subclasses. This allows us to define our systems in terms of many interchangeable parts.

Let’s say that we have a Dog and a Cat and a Turtle class, all of which are subclasses of Pet. If we are following the Liskov Substitution Principle, we should be able to replace any usage of Dog, Cat, or Turtle in our code with Pet and have our code still work.

This allows us to write any code in terms of Pet, rather than using specific subclasses. This code is more extensible because any future Pets that follow the Liskov Substitution Principle can be added and will work with our codebase without needing to change any existing code. This also furthers our use of the Open-Closed Principle.

**Interface Segregation Principle**

**If we have written a lot of object-oriented code, we have probably had to implement methods that we don’t actually plan on using**. We sometimes need to add these methods in order to make a class conform to an interface, a description of the inherited properties and methods that need to be implemented by a subclass, we need to use.

When we decide to add methods to an interface that are only used by some of the subclasses, we are violating the Interface Segregation Principle. We should instead define new interfaces that properly abstract the different types of subclasses.

**Dependency Inversion Principle**

While the Liskov Substitution Principle has us design classes that can be substituted for their superclass in client code, the Dependency Inversion Principle has us actually write our client in terms of those superclasses.

The Dependency Inversion Principle states **that systems are the most flexible when dependencies (classes calling other classes) refer to abstractions rather than concrete classes**. This ensures that we define appropriate high-level interfaces and write code that is generic to those interfaces, enhancing our use of the Open-Closed Principle as well as the Interface Segregation Principle.

**Review**

**Which is NOT a summary of a SOLID principle?**

* A system should be built out of a small number of large classes. This would likely cause a violation of the Single Responsibility Principle.

In this article, we have introduced the SOLID principles for creating software that is easy to change, understand, and reuse. Rather than being independent guidelines, the SOLID principles support and reinforce each other, resulting in cohesive guidelines for well-designed systems. Recognizing violations of these principles in our codebases and fixing them is an important aspect of improving the health of our system designs.

**Introduction to Design Patterns**

**This article introduces the concept of design patterns, their main categories, and provides brief examples of how they can be used within a development effort.**

At the beginning of our project, we could add new classes and features without much hassle. But as our project grew bigger, it kept getting harder to make updates. We forgot to think through the design of our system. We needed some help getting back on track.

This article is here to help with design patterns, which are reusable solutions to problems that commonly occur during software development having to do with the way we organize our components, the pieces of code that make up our application. Design patterns can be thought of as a set of ways to structure system components and how they communicate with each other.

**Why Do We Need Design Patterns?**

Software continues to be worked on far after it is released. Most of the development time is spent modifying or maintaining a released software system. A software system change can result from:

* Adding new code
* Changing existing code to work with the new code
* Deleting code

Over time, these changes complicate our system, making it difficult to make more changes. Design patterns allow us to engineer our systems to better accommodate change. We often will pick a pattern to implement based on the problem that we are trying to solve. As we practice with design patterns, we develop our ability to identify the pattern that should be applied to the problem we are seeing in our system, as well as anticipating where these problems will occur.

**Types of Design Patterns**

There were three major types of object-oriented design patterns first proposed. Over time, more types of design patterns have been introduced, but here we will focus on those most applicable to any object-oriented system.

* **Creational patterns** hide the logic needed to create an object from clients that need to use the object.
* **Structural patterns** provide ways to combine objects into larger structures.
* **Behavioral patterns** define clear ways for objects to interact with each other.

**Creational Patterns**

**Creational patterns** solve problems that involve hiding the logic required to make objects from components that use the objects.

Let’s say that we have a class that interacts with a database. When we connect to the database, we need to pass in credentials and configurations. Should every class that needs to access the database need to pass all of this information in? Does the setup process need to happen each time the database is first accessed by an object?

The Singleton pattern is an example of a Creation pattern that solves this creation conundrum. This pattern is often used as the way to access an application’s database or anything else that might not need to be instantiated more than once. The Singleton pattern gives a codebase widespread access to the same singular instance of an object. This pattern enforces that only one instance exists and prevents additional instances from being created.

**Structural Patterns**

**Structural patterns** are all about composing classes and objects into larger structures.

Imagine we are creating a hierarchy of objects in which there are many variants. Our Rectangle might have RedRectangles, BlueRectangles, RoundedRectangles, RoundedBlueRectangles, and all sorts of other types. Each time we add a new feature of a Rectangle the number of classes needed to represent every variant explodes combinatorially. Instead of representing each variant as a class, we need to create classes out of types of these variants and have our base object use these as instance variables.

One specific example of a Structural pattern is the Bridge pattern. This pattern seeks to replace some of the responsibility of inheritance (many subclasses) with object composition (making objects out of other types of objects). We seek to take a dimension of an object that is causing the number of subclasses to grow, turn that dimension into a class, and make an instance of that class an attribute of the base object.

**Behavioral Patterns**

**Behavioral patterns** are those that don’t fit into the Creational or Structural pattern categories. These patterns are described as defining ways for objects to interact with each other in an extensible way.

For example, imagine we want to be notified when certain events happen to one of our objects, such as when a field is changed. The Observer pattern, a specific Behavioral pattern, allows us to add new entities listening in on these events and create new events that these objects can listen to.

**Which type of design pattern is concerned with hiding the instantiation logic of a class?**

* Creational patterns are concerned with hiding the logic of object creation or instantiation.

**When Should Design Patterns Be Used?**

When first learning design patterns, it might be tempting to apply them throughout our systems. However, too many design patterns are likely to overcomplicate things. Design patterns are best applied in two scenarios:

When we are encountering a problem that a design pattern is meant to solve

When we are introducing a part of our system that is likely to change repeatedly

For example, if we are struggling to come up with a design for how two pieces of our system might communicate with each other, we may want to look at the problem descriptions for behavioral patterns. You might find one that applies directly to the situation we are in. This would be the perfect time to apply that pattern.

Throughout this article, we have introduced design patterns, the categories in which they are organized, and described the scenarios in which several important design patterns are used. Design patterns give us solutions to common object-oriented problematic situations, as well as a shared vocabulary for discussing design decisions.

**Which statement is NOT true?**

* Design patterns are best chosen before development begins. Design patterns are best chosen during development in response to a problem that is occurring.

**Introduction to Model View Controller**

**This article introduces the Model View Controller design pattern.**

As we design our software we sometimes need to view and organize our system at the big picture level. One popular architectural pattern is the Model View Controller pattern, or MVC. Throughout this article, we will discuss the use of MVC to organize the main components of our systems by their functionality.

**Model View Controller**

Model View Controller is an organizational design pattern used for applications that need:

* To move around data throughout the application.
* To display the visual and interactive elements to the users of the application.
* A means of having the user’s actions change the application data.

These needs are so common which is what makes MVC such a great pattern to start the design of applications! The Model View Controller pattern organizes a system into three types of components: the Model, View, and Controller.

Imagine we are designing a website that displays a fun fact each time a button is clicked. Let’s discuss how we would organize the site with the Model View Controller pattern.

**The Model**

The Model is made up of the data storage, as well as any classes that represent that data as it moves around the application. Data is often stored outside the application in a database or files. While the data can be stored in different formats, we often read data from storage into a representational object. These objects should be extremely simple and have very little behavior outside of allowing access to their data.

Within our “fun fact” site, our model might consist of the storage of our facts, as well as the classes that represent them.

**The View**

The View component is the classes that describe how our application will be presented to the user — it’s what the users see. These might be our React components or HTML elements in a web application or XML files in an Android application.

In our fun fact site, this would be the code that displays the button to click and the currently visible fact.

**The Controller**

The Controller is the brain of our application. The View and the Model do not define much behavior for our applications, instead, they merely represent presentational and data objects. The Controller defines the behaviors that our system will accomplish using the Model and View. The Controller is responsible for receiving events (clicks, submitted forms, typing) passed in from the View and processing them to make meaningful responses. The Controller will interact with the Model, making queries or representing data as appropriate to make these responses happen.

In our example application, the Controller of our fun fact website would receive the button click event from the View, and request a new fact from the model, then send that new fact back to the View for display.

**Which category of MVC is responsible for representing the data of our application?**

* Model

**Benefits and Disadvantages of MVC**

The primary advantage of the MVC pattern is the separation of the data representation, logical, and presentational layers. By keeping these aspects from being highly tied together, they can be modified independently. Changing the Model should not require major changes to the View, and vice versa.

By separating the Controller from the View, we can create multiple ways of viewing our application. For example, an application can have the web and mobile Views interact with the same Controller. This can allow us to greatly reduce the amount of work needed to port our application to new means of user interaction.

The main drawback of MVC is that it can introduce unneeded complexity to an application. Having multiple components and structure may not be necessary for simpler applications. A decent guideline is that if our application requires multiple people for development, having a pattern such as MVC probably would be helpful. This will help team members reason about class intents and where an object might fit into the application.

Overall, the MVC pattern is ideal for more complex projects in which there are expected to be changes in the way data is represented or presented over the product’s lifetime. The separation of these concerns allows flexibility when making changes to the system.

**Which is NOT a benefit of using the MVC pattern?**

* It simplifies the structure of a basic application. MVC might be too complex for a basic application.

**Review**

In this article, we introduced the Model View Controller pattern for organizing our system at a high level. This pattern separates the following aspects of our application:

* The way we store and represent our application’s data, **the Model**
* The way we present our application to the user, **the View**
* The way our application achieves system behaviors, **the Controller**

**MVC** is an important and popular way to design applications and a crucial part of the software designer’s tools. Consider using the Model View Controller pattern the next time you build a complex application!

If you are interested in learning more about these topics, here are some additional resources:

* [UML Official Site](https://www.uml.org/)
* [Design Patterns by Refactoring Guru](https://refactoring.guru/design-patterns)
* [Clean Coders: Software Architecture](https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html)