

Throughout this reading, we will attempt to:

- Define software architecture as it relates to Software Engineering and its importance.
- Discuss what makes an architecture good.
- Discuss the SOLID principles for developing a modifiable architecture.
- Discuss how to begin developing an architecture for our projects using the Model View Controller pattern.
- Discuss some appropriate views through which to understand our software architectures.

What is Software Architecture?

Software Architecture in Practice states:

“The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.”

Software architecture in other words, allows us to hold in our heads a model of our system, to be able to break it into pieces or components, and understand how those pieces fit together at various levels of abstraction.

Software Architecture in Practice makes the important point that our functionality does not determine the software architecture we are going to use. There are unlimited ways that some functionality can be encoded into software components through the use of functions, classes, methods, and various other structural tools.

As we start our project, we will find it incredibly easy to just add new functionality in any which way, without even thinking about software architecture. In Agile development, we are constantly focusing on writing code that provides customer value. We are looking to constantly provide bits of end to end functionality. A little of the database here, some networking there, and some user interface elements here. No where in the Agile manifesto or Scrum guide does it prescribe how we should organize this implementation.

When implementing many different aspects of the system all the time, it becomes all too easy to add them in an ad hoc fashion, taping together dozens or hundreds of different components.

However, each of these “ad hoc” additions drastically increases the complexity and the chaos within our system.

Around the time we get our first or second points of feedback, and it is time to change or add new things to the system, you will find that making changes is no longer easy when your system is an unintelligible spiderweb of components and responsibilities. When this happens, teams either have to significantly rework their code, or sometimes even start over.

In the last reading, we described how Git provides us with an essential means for sharing our code changes amongst our team. A good software organization provides us with an essential means for both understanding how our system works, and a framework for organizing where those code changes belong.

Especially important to us at the beginning of a project, developing a software architecture causes us to put down some of our earliest and most important decisions about how we think our software will work.

Some of these decisions are:

- How will the major portions of our software be organized?
- How will those portions communicate with each other? What information needs to be conveyed in that communication process?
- Does data need to be persisted between uses of the application? What will the structure of that data be? How will that data move throughout the application?
- How and where will external APIs or tools connect to our application?

One problem is that we can answer all of these questions and still produce a bad architecture, so let's take a look at some guidelines for producing an architecture of a high quality.

What Makes a Software Architecture Good?

Practices of a good architecture from *Software Architecture in Practice*:

- Our architecture lends itself well to incremental additions without vastly increasing system complexity.
- The architecture is documented using the views that we will discuss later in this reading.
- Each part of our system should have a well defined purpose.
- The things most likely to change should use information hiding and encapsulation, preventing other areas of the system from having to change along with it.
- Modules should have a well defined interface that encapsulates changeable aspects of the module from others.
- When practical, your architecture should be composed of well defined design patterns and tactics. Some of the most important of these will be defined later in this reading.
- Parts that create data for your system should be kept separate from the modules that consume that data and enter it into the system.

- The architecture should feature a small number of ways for components to interact. That is, the system should do the same things in the same way throughout.

With these in mind let's explore some tactics which will help ensure we create modifiable and usable software.

Creating a Modifiable Architecture

In *Clean Architecture*, Robert Martin explains his SOLID principles for creating software modules that:

- Tolerate change.
- Are easy to understand.
- Are the basis of components that can be used in many software systems.

According to Martin, "The SOLID principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected."

SOLID is an acronym that stands for:

- SRP: The Single Responsibility Principle: Modules should have one, and only one reason to change.
- OCP: The Open-Closed Principle: "For software systems to be easy to change, they must be designed to allow the behavior of those systems to be changed by adding new code, rather than changing existing code."
- LSP: The Liskov Substitution Principle: Software systems should be built out of interchangeable parts, and for this to happen "parts must adhere to a contract that allows those parts to be substituted for one another".
- ISP: The Interface Segregation Principle: Modules should not depend on anything they don't use.
- DIP: The Dependency Inversion Principle: "Code that implements high level policy should not depend on code that implements low level details. Rather, details should depend on policies."

While we will provide a high level overview of each of these principles, in order to fully grasp them you will likely need to do additional reading.

The Single Responsibility Principle:

This principle states that a module should have exactly one reason to change. The definition of Module that Martin means here is generally "a single source file", or more generally "a set of cohesive functions and data structures".

More important is what is meant by "reason to change". This is easiest to understand by example.

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, stored in database on save.

At one level, it seems our class follows the Single Responsibility principle, if something about the Enemy changes, then this class has to change.

But what are the main reasons that the Enemy class might change?

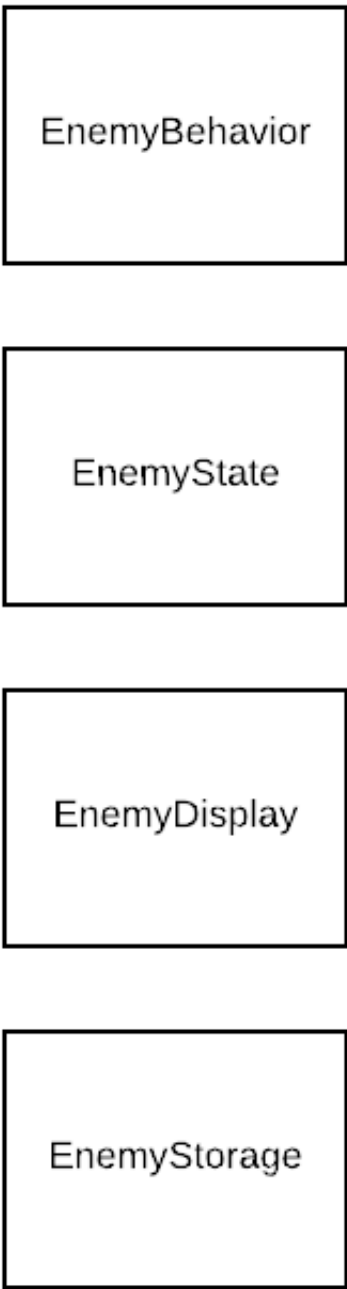
The mechanisms of the gameplay might change, in which case the behavior of the enemy (attacking and taking damage) would be affected without necessarily changing up how an enemy is displayed on the screen or changing any of the state information about the enemy.

The game's artistic direction might change, necessitating changes to how the enemy is displayed on the screen without changing the enemy's behavior or state information.

Finally the database administrator might require changes to how the state information of an enemy is stored, without changing either of the other two responsibilities.

It becomes clear that there are actually several reasons for this class to change. If we feel that these reasons best represent these method's reasons to change we should split the class into the following classes:

- EnemyBehavior: Handles the 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.



```
graph TD; EB[EnemyBehavior]; ES[EnemyState]; ED[EnemyDisplay]; EST[EnemyStorage];
```

EnemyBehavior

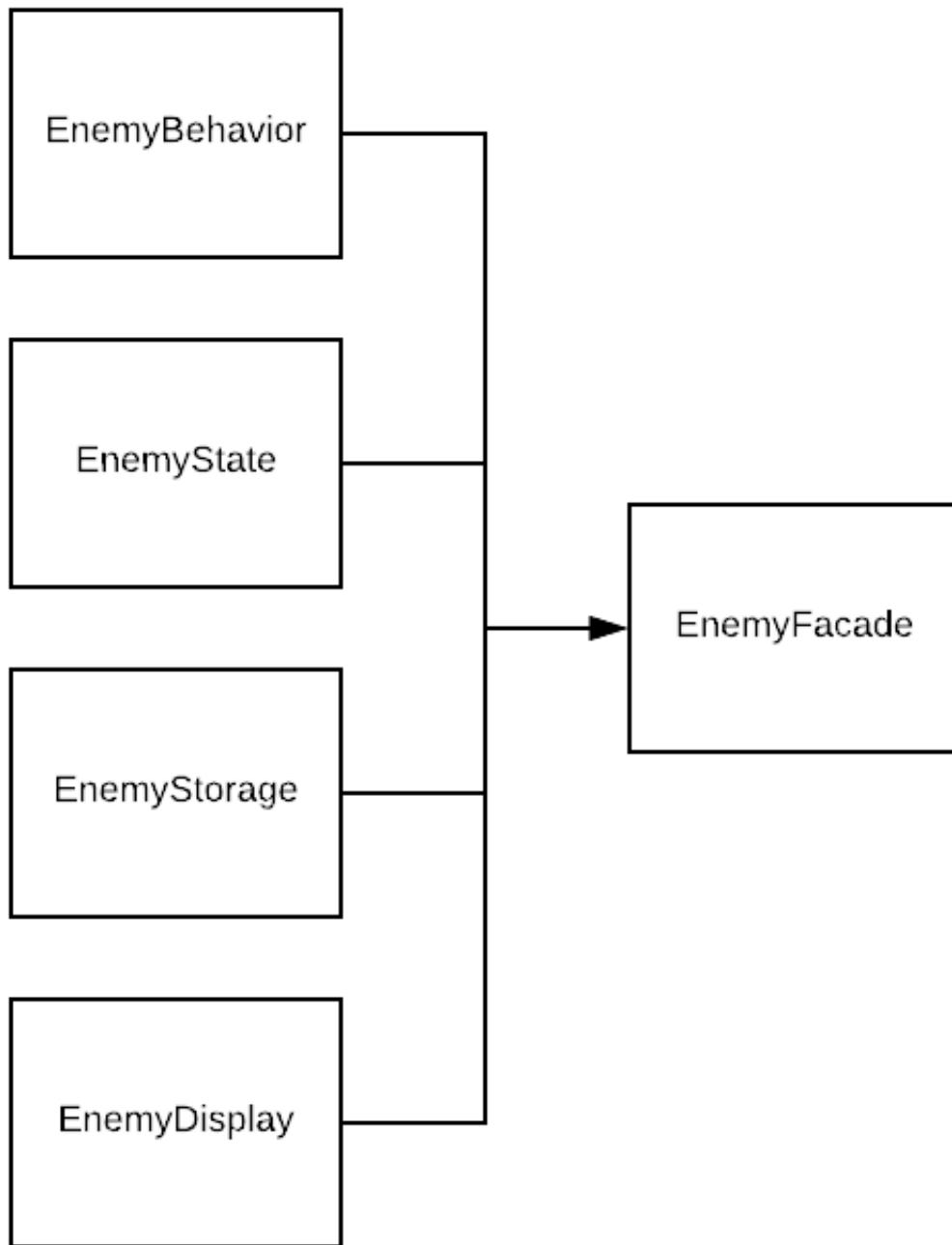
EnemyState

EnemyDisplay

EnemyStorage

One obviously apparent down side is that we now have 4 classes for Enemies that we have to instantiate and keep track of. This is the explicit purpose of the Facade Design Pattern.

This involves the establishes of a fifth class, EnemyFacade, whose sole purpose is to instantiate delegate access to the other Enemy classes.



Classes outside of the Enemy group will interact with the EnemyFacade rather than any of the new Single Responsibility classes directly. As long as the Single Responsibility classes follow a predefined interface, the previously discussed changes will not affect the EnemyFacade class, making it also have a Single Responsibility.

More information on the Single Responsibility Principle:

<https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleResponsibilityPrinciple.html>

The Open Closed Principle

A change to a software system has typically two parts:

- The new code that actually implements the change in behavior we want.
- The changes to our existing code that accommodates that new behavior.

Students in Software Engineering tend to have a relatively small number of classes, and as changes come into their system, they typically have to modify one or more of these classes to accommodate the change. As these classes get more and more bloated and change filled, they become harder and harder to change. While the former element might be relatively easy, the latter element might still stop the change from happening.

Ideally, adding a new behavior to our system only consists of doing the former element. A good software architecture minimizes the latter element, and allows us to make new behavior without changing our existing code.

The Open Closed Principle is defined as:

A software artifact should be open for extension but closed for modification.

Example

In *Clean Architecture*, Martin asks us to:

Imagine that we have a system that displays a financial summary on a web page, but our stakeholders ask that this same information be turned into a printed report.

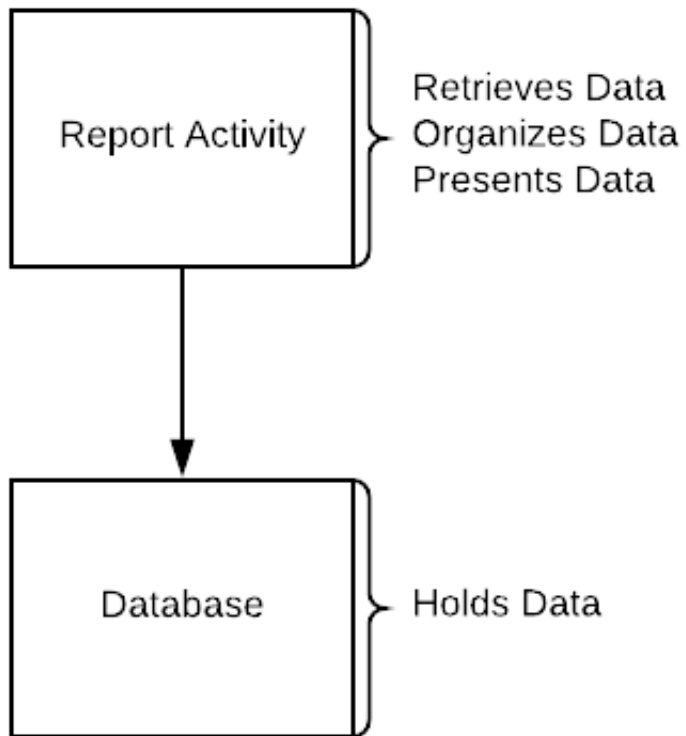
In order to implement this change, some new code will have to be written, but how much of the old code will have to be changed?

What you should be thinking is: “Since producing a web page has absolutely nothing to do with making a pdf, we shouldn’t have to change any code at all.”

That would be absolutely right. But this is only true if we have separated the following processes:

- Getting the data from the database
- Turning that data into a report
- Turning that report into a web page

When these processes are combined into mixed together classes, we will have to edit the existing class in order to add our change. A typical Software Engineering team might produce the following solution:

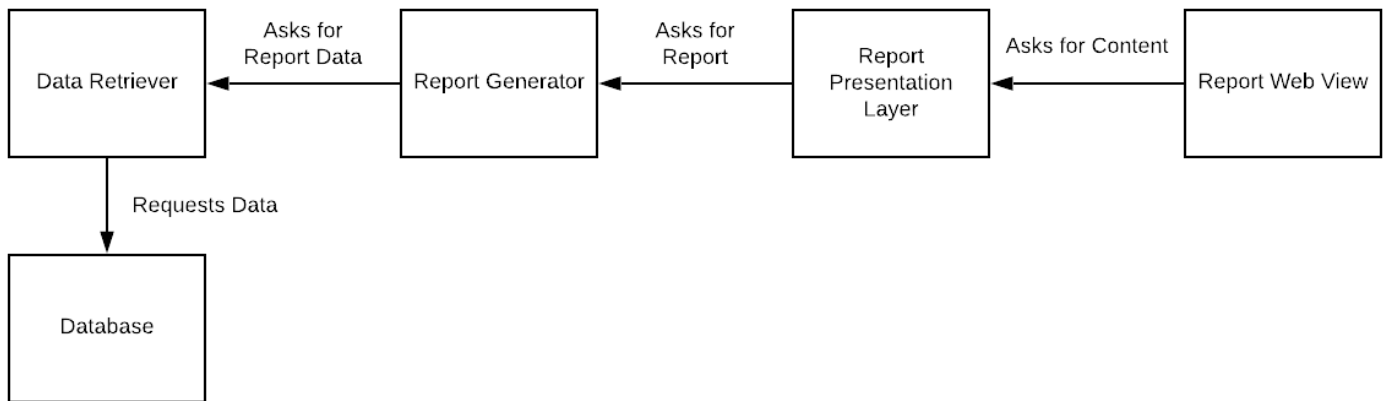


This ReportActivity would need to change for the following reasons:

- The database model for the report changed
- The necessary data for the presented report changed
- The layout of the presented report changed

Therefore not only does this architecture violate the OCP, but also the SRP as well.

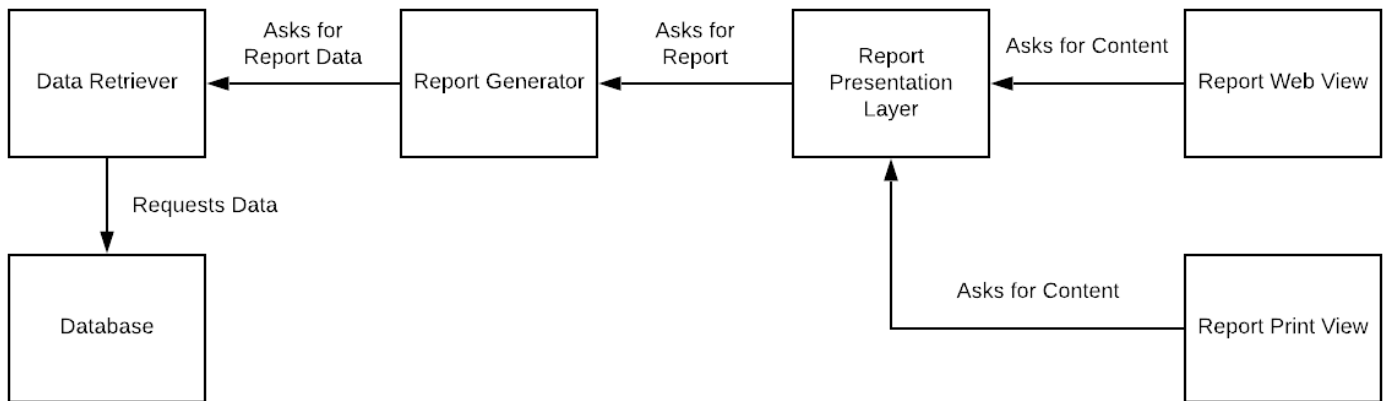
A much more modifiable solution would be the following:



- Database changes now only affect the Database Retriever.
- Changes to the presented report content affect only the Report Generator.
- Changes to the layout of the presented report only affect the Report Presentation layer.

This architecture therefore satisfies the SRP and the OCP.

When we need to add the Printed Version we simply add a new Report Printed View class.



When using the Open Closed principle we are always looking to "separate functionality based on how, why, and when it changes, and then organize that separated functionality into components".

The Liskov Substitution Principle:

This principle is a way of defining subclasses to be interchangeable with each other.

The point of having subclasses is that we can write code at a high level that only deals with the superclasses. When we need new functionality, we can implement it as a subclass of something that already exists, and the

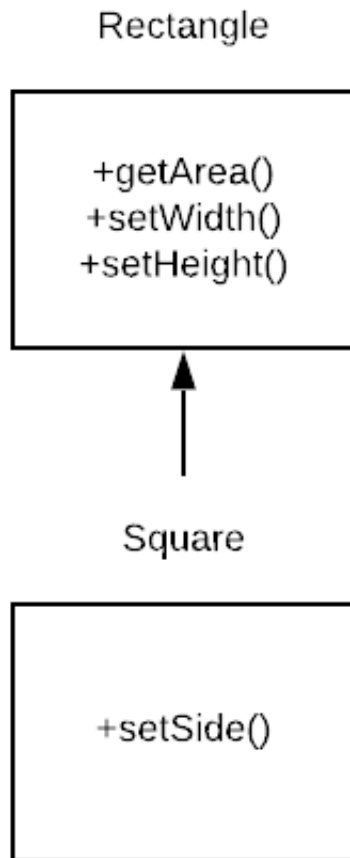
high level code does not need to change in order to process it.

This allows us to write highly generalized code, which can handle infinitely many additional subtypes.

The Liskov Substitution Principle essentially states that a program that uses a superclass should be able to replace that superclass with any of its subclasses and have the program still work as intended.

This is basically saying that if we have a good subclass, it should be able to be used as its superclass without the user knowing of the details of the subclass.

In order to understand this let's go through an example that is a violation of the LSP:



```
int resizedArea(Rectangle rect) {
    rect.setWidth(2);
    rect.setHeight(5);
    return rect.area();
}
```

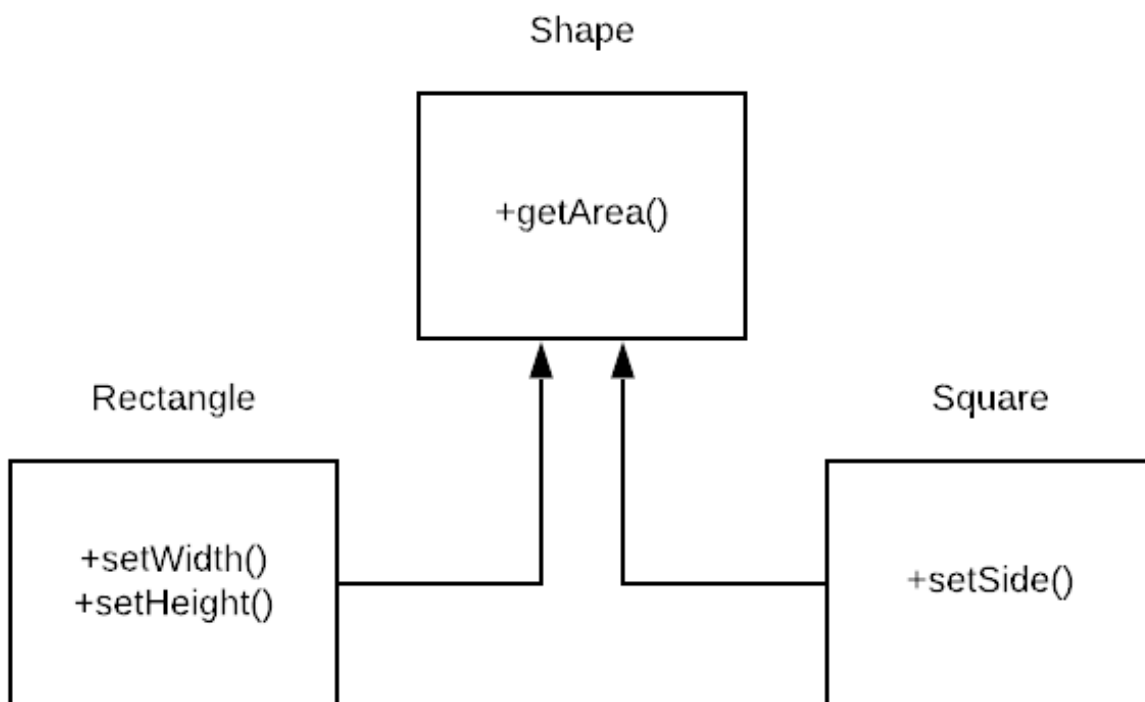
In this example, Square is not a proper subclass of Rectangle because while width and height are independently mutable with Rectangle, this is not the case with Square, in which changing height would change width and vice versa.

Users, thinking they are independently changing the size of the sides, will actually be changing both when Squares are passed in.

This could be protected against by adding logic that checks for Square instances, but each time we do this we are increasing the complexity of the system.

As we add more subclasses and more checks, our system loses maintainability and violates the Open Closed Principle.

We instead want to design our superclasses and corresponding subclasses in such a way that our usage of the subclass does not depend on any of its details.



There is now no way to use the superclass in a way that depends on the subclass details.

Interface Segregation Principle:

If you have ever written a substantial amount of Object Oriented code, you have probably had to implement methods that you don't actually plan on using in order to make a class conform to an interface you need to use.

These methods might not be implemented for the following two main reasons:

- Your object doesn't need to do that behavior.
- You don't know what these methods are supposed to do or what they are for.

Often times this is going to occur when you are using an object within a framework, such as an Android Activity or a method defined by a library,

This problem is somewhat unavoidable when dealing with other company's interfaces, but the Interface Segregation Principle is about preventing this problem with the interfaces you design.

This problem doesn't necessarily occur when first designing the interfaces. Typically people don't set out designing extra methods on interfaces that their objects aren't going to use.

What happens is that if we are using the other SOLID principles appropriately, we are constantly adding new subclasses that implement this interface.

Eventually we might discover new functionality that several of these subclasses do in common that are not in the original interface. This is where violations of the Interface Segregation Principle are most likely to occur.

When we decide to add methods to the interface that are only used by a subset of the subclasses we are violating the ISP, the SRP, and losing cohesion amongst our classes.

Instead, we should seek to define new interfaces that properly abstract the different types of subclasses.

Overall, we should take notice when we are not using methods defined by interfaces that we created. This is often a sign that our abstractions are not properly defined and we should reevaluate our interfaces.

Most students are relatively new to OOP, and tend to be reluctant to use abundant amounts of interfaces, so we will not go into much more detail on this principle.

For more detail and an extended code example go to:

<https://stackify.com/interface-segregation-principle/>

Dependency Inversion Principle:

Essentially, this principle states that systems are the most flexible when dependencies (classes calling other classes) refer to abstractions rather than concrete classes.

In Java, abstractions would be interfaces and abstract classes whereas concrete classes are typical objects with their implementations filled in.

What Martin is saying here is a pretty radical departure from how we have typically approached Object Oriented Programming, that we should not be referencing any of our own concrete classes from any other class, only abstract classes. This does not refer to standard library objects such as String or ArrayList.

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.

To follow this principle Martin recommends the following coding practices:

- Don't refer to your system's concrete classes from another concrete class.
- Don't derive from your system's concrete classes.
- Don't override concrete functions.
- Using the Abstract Factory design pattern to instantiate new object instances.

This can be a rather daunting task for beginning software engineers, and it should be kept in mind that this is a principle rather than a rule.

For more information on the Dependency Inversion Principle and an extended coding example go to:

<https://stackify.com/dependency-inversion-principle/>

These SOLID principles intend to help us keep our systems modifiable, but how do we get started in the first place? There is one design pattern that is a great starting place for almost any user facing application, the Model View Controller pattern.

Getting Started With Model View Controller:

At this point you might feel overwhelmed by all this talk of interfaces and design patterns, and are just wanting to know, "How do I get started designing my system?"

Luckily, there is a simple design pattern called "Model View Controller" or MVC that is a great starting point for almost any application that:

- Has to move around data throughout the application.
- Has to display the data and interface elements in some way to the users of the application.
- Needs a means of understanding the user's input and making relevant changes to the applications data.

You might have noticed that those three sentences could apply to almost any application that has a user. This is what makes MVC such a great pattern to start the design of applications we create!

Jeff Atwood, Creator of StackOverflow.com defines Model, View, and Controller as the following:

Model

The classes which are used to store and manipulate state.

View

The user interface bits necessary to render the model to the user.

Controller

The brains of the application. The controller decides what the user's input was, how the model needs to change as a result of that input, and which resulting view should be used.

Let's break each of these down a little further:

Model:

As Atwood said, the parts which make up the model are “the classes which are used to store and manipulate state”

Some people get this confused with the Database of the application, or the class that accesses the Database.

While the Database can be considered part of the Model, it is not all of it, and the class that accesses the Database is actually part of the Controller.

We would like to distinguish between two types of data used in our applications, persistent and non persistent data.

Persistent data is data that is retrievable even after the application has been closed.

This type of data needs to be stored outside the application, typically in a database or files. They are also not stored in a format that is directly equivalent to an object in our system.

Take for example the following JSON student record:

Nick.json:

```
{  
  Name: "Nick",  
  Age: 25,  
  Id:"00110321",  
  Major:"Computer Science"  
}
```

When we read this from our file system, or we query from our database, we are getting the persistent data and putting it into our system.

The data in our system is only a copy, and unless saved back into our database or file system, will be lost. This is non-persistent data.

Non-persistent data does not only come from database calls but can also be created as a result of user inputs or our system's processes.

Often times, in order to efficiently move this non-persistent data around our system in a way that is easily usable, we translate it into an object that represents an abstraction of what that data means. For the student data example, that object might look like:

```
class Student {  
    public String name;  
    public int age;  
    public String major;  
    public String id;  
}
```

The combination of the persistent data storage, as well as any classes we are using to abstract the data moving around our application is what makes up our systems Model.

These objects that are within the Model should be extremely simple and have very little in terms of behavior outside of allowing access to their data.

As we will explain later the behaviors involving these Model classes are defined by classes within the Controller.

View:

The View is the means by which we show the user how to interact with our model.

In web development, the View is made up of the HTML and CSS, and in Android, it is made up of the XML, and Fragments.

We now have the means by which we represent our data, and by which we present our screens to the user, the Model and the View. However the objects are empty, the database unread, and the screens unconnected from our Model.

Controller:

The Controller holds a wide variety of functions that, if it were all in one module, would clearly violate most of the SOLID principles. It is instead made up of the majority of the classes in our system.

The Controller:

- Reads information from the database
- Converts it into our Model objects
- Passes the objects through the various system processes
- Structures the display of these objects in the View
- Receives the user input from the View
- Makes any necessary changes to the persistent and non persistent data

There is some debate over whether Activities count as part of the View or as part of the Controller in Android. In either case, Activities should be as small as possible with most of the Controlling logic happening in other classes.

Students often cram many of these functionalities into their Activities, making their Android Activities overly large, complicated, and difficult to change.

It is during the development of the Controller that the SOLID principles are most important to maintain modifiability.

Getting Started:

The MVC pattern provides us with a useful entry point into building our applications.

To get started, we look at the most important user stories, and as a team:

- Discuss the View components necessary to have the user achieve that story.
- Discuss the Model components that can represent the data used throughout the Story.
- Discuss the Controller components that connect the Model and View components as well as implement the necessary backend processes that make the user story work.

Each of these components will be the tasks that we work on throughout the Sprint.

More Information:

For more information on the MVC in Android:

<https://android.jelise.eu/android-architecture-2f12e1c7d4db>

In Unity:

<https://www.toptal.com/unity-unity3d/unity-with-mvc-how-to-level-up-your-game-development>

In Swift:

<https://www.appcoda.com/mvvm-vs-mvc/>

It is worth noting that with SwiftUI, apparently the dominant architecture is not MVC, but MVVM for “Model, View, ViewModel”.

Architectural Views of Software (Not Part of MVC)

We have now discussed a variety of practices for developing a good architecture, as well as a way to get started. You may be wondering how we are going to keep track of all of these software components and how they relate to each other.

A complex software system cannot be understood from a single viewpoint. It is only through a variety of views that we can achieve a comprehensive understanding of the system.

In this section we will discuss a variety of such views through which we can understand our systems.

Code Components

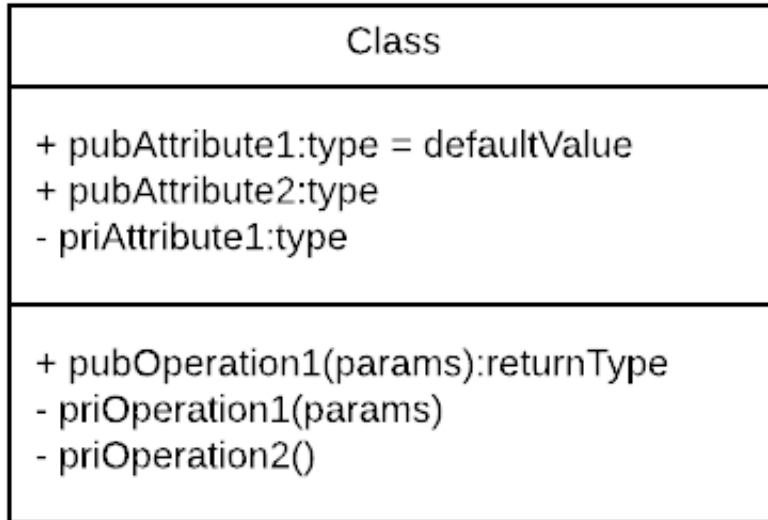
In order to understand how our code works, we should not need to read it line by line. Through properly named class names, methods, and attributes we should be able to understand the purpose of an object without seeing its code.

In systems with many classes, it can even be overwhelming to look at this reduced amount of information. We therefore present three different levels of representing code components and the purpose of each level.

Low Level

At the lowest level is the typical UML Class diagram. It displays:

- The class name
- Its attributes
- Its private and public methods

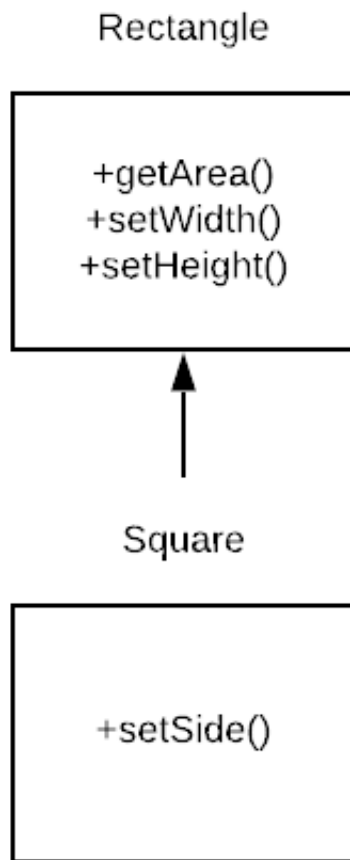


As you know from implementing these in previous programming courses, these can be very useful in understanding how a class is going to work, and how it will be implemented.

However, when trying to get the bigger picture, all of these details can be unnecessary noise.

Medium Level

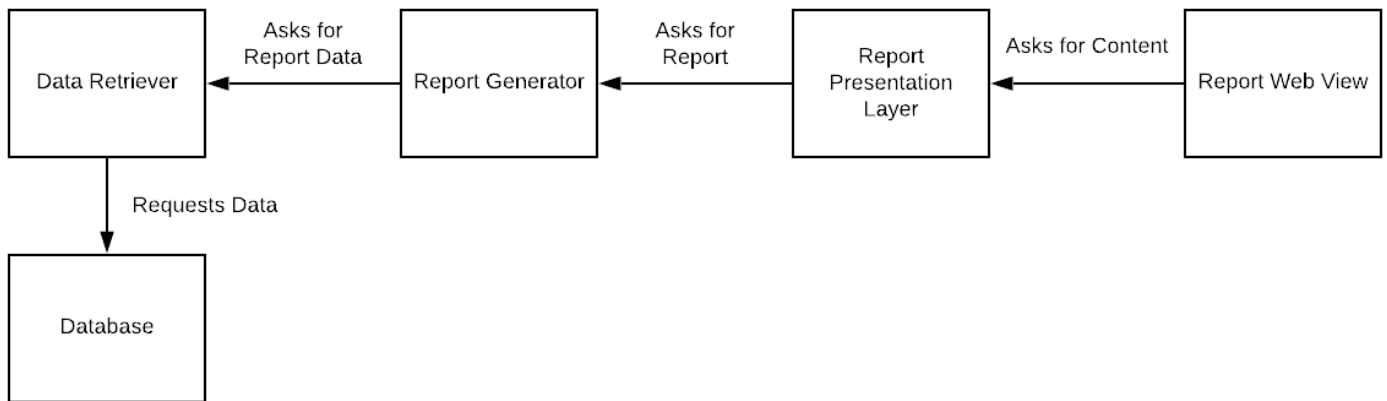
At the medium level of abstraction, many of the details of the class are removed, instead only public methods and attributes are visible.



This level of view can be useful when designing how a class will interface with other classes.

High Level

At the highest level, a class is just reduced to its name and the connections it has to other classes.



This level of abstraction can be particularly useful when trying to understand how various modules come together to form a part of the system.

Databases

As previously mentioned, databases do not store their information in the objects that we use in our system. Their structure therefore needs a different means of communication for code components.

Databases consist of many tables, with each table holding records about particular entities. These entities will often have relationships to other entities throughout the database, whether in the same table or not. Each record will consist of one or more columns indicating some aspect of the entities' data.

Data Dictionary

When initially designing our databases, it is important the the team is in agreement about what a particular table holds, and what each attribute is made of and what it is for. This is the purpose of a data dictionary.

Event	An event that may occur when a new room is discovered, which the player must roll a number of dice - dependent on	Event ID
		+ Flavor Text + Action Stat + Thresholds + Result Stat + Result Value

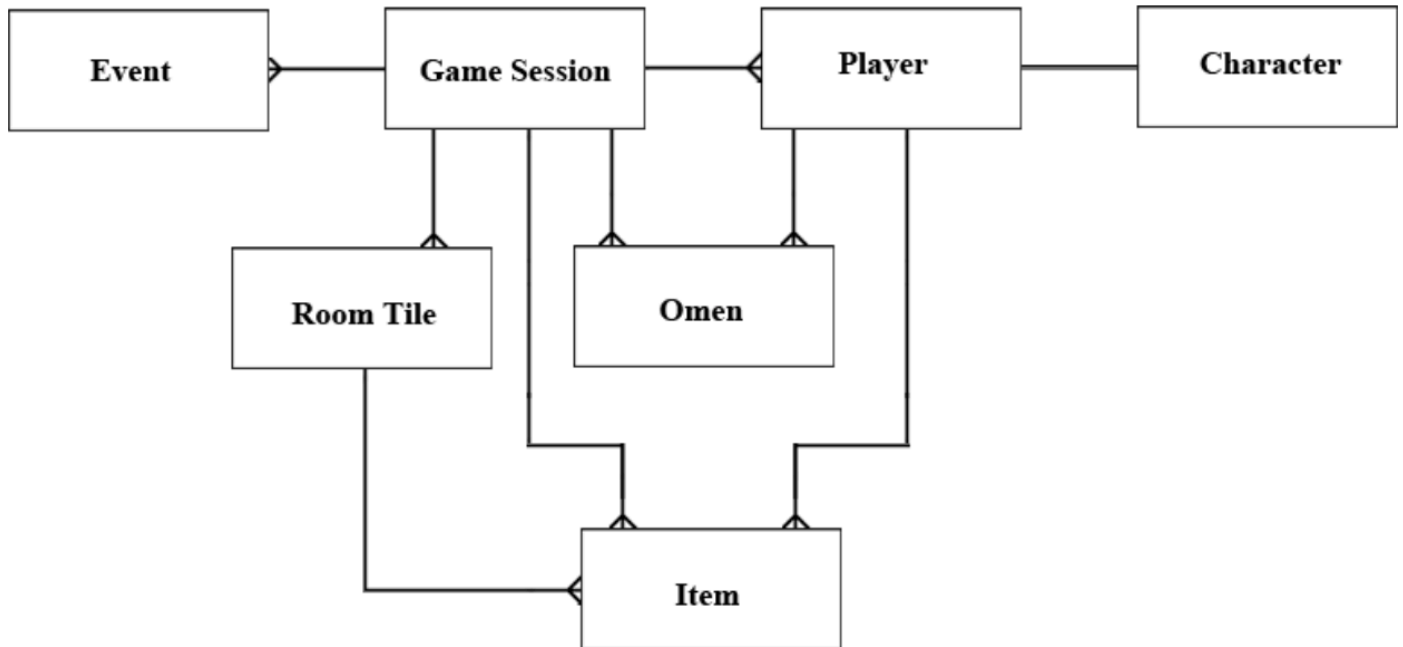
	one of their stat values - to try to resolve. May increase or decrease the player's stat(s)	
Thresholds	Describes the dice roll values that define each level of success for the event: Failure, Mixed Success, and Success	Integer array
Action Stat	The stat that the player must use in resolving the event, which determines the number of dice that the player will roll	String
Result Stat	The stat that the player will modify according to the results of their	String

roll

Example Portion of a Data Dictionary from a student capstone project

ERD

An ERD takes a larger view of the database system, showing mainly the types of entities and their relationships to one another. These relationships are often shown using a “Crow’s Foot Notation”.



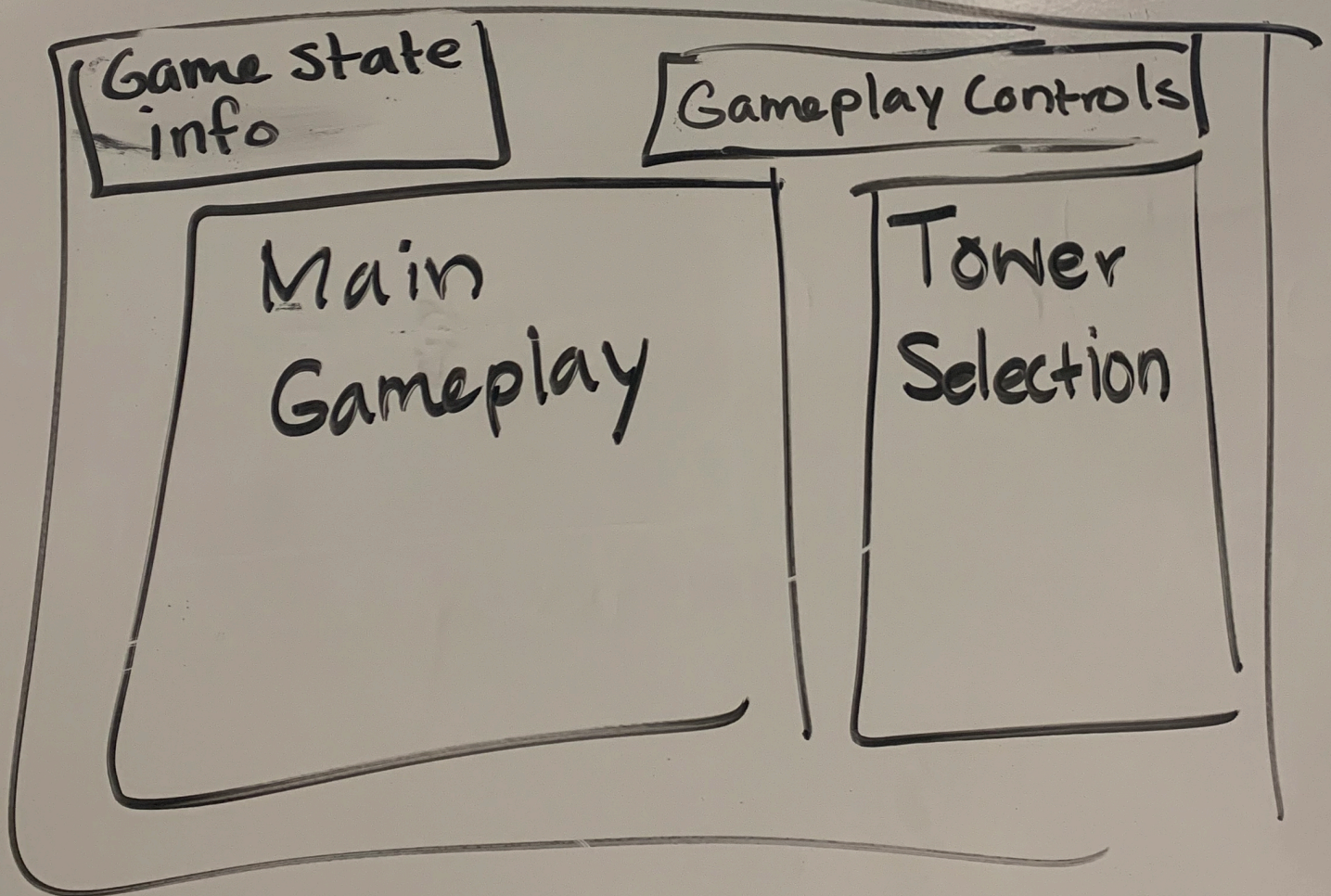
For more information on ERDs and Crow's Foot Notation go to:

<http://www2.cs.uregina.ca/~bernatja/crowsfoot.html>

UI

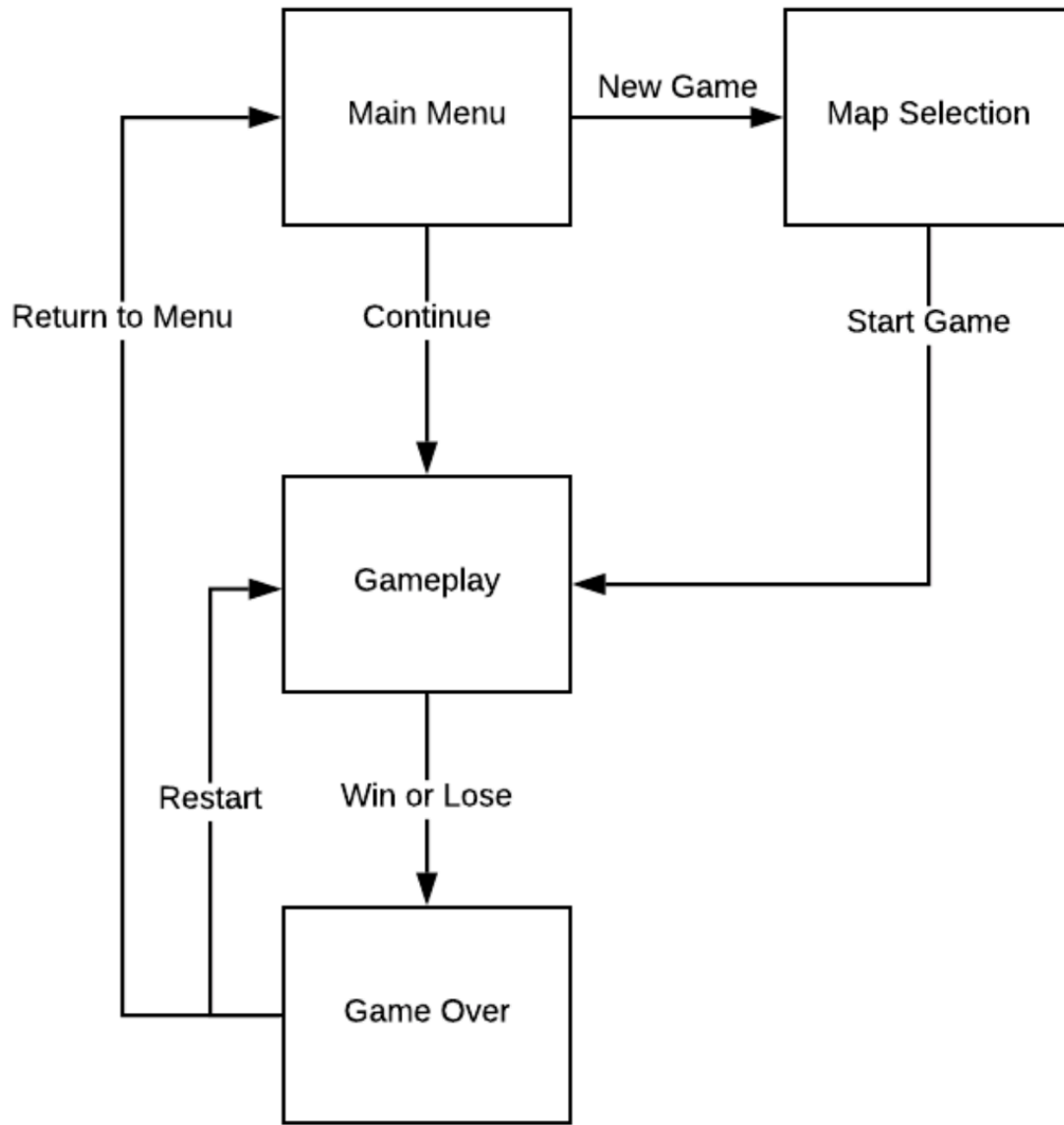
In the previous reading, we have already partly discussed the means of conveying what information and interaction elements will be on the page, through the use of Wireframes. These can represent an important simplified view of our screens to assess whether all of the needs of our user stories are being met.

Main Game Screen



Map

When designing an application, we often want to know how our user will be “flowing” through the screens to accomplish their activities. In order to visualize this, a map of our screens showing a representation of each screen and how they connect can be extremely useful.



At the moment, our application shouldn't consist of many screens, but as it grows, this type of view may become more useful.

Summary

Throughout this reading we have:

- Defined software architecture and its importance.
- Discussed what makes an architecture good.
- Discussed the SOLID principles for developing a modifiable architecture.
- Discussed how to begin developing an architecture for our projects using the Model View Controller

pattern.

- Discussed some appropriate views through which to understand our software architectures.

This is a lot of information, and cannot be mastered all at once. This reading is meant to serve as a guide all throughout the development process as you continuously make architecture design decisions.

In the next reading, we will cover the remaining information for planning and working throughout our sprints.