Module 3

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 3 Study Guide and Deliverables

Module Topics:	• Topic 1: High-Level Design – Software Architecture • Topic 2: Design Principles and Design Patterns
Readings:	• Online lecture notes • Braude, Part V (or related chapters in other textbooks)
Discussions:	• Weekly Group Meeting
Assignments:	• Project Iteration 1 due **Tuesday, September 28 at 6:00 AM ET** • Post-Project Iteration 1 Review due **Tuesday, September 28 at 6:00 AM ET** • Weekly Report due **Tuesday, September 28 at 6:00 AM ET**
Live Classrooms:	• **Tuesday, September 21 from 7:00-9:00 PM ET** • **Saturday, September 25 from 8:00-9:00 PM ET**

Learning Outcomes

By the end of this module, you will be able to:

- Describe and compare different design goals
- Explain the definition and the importance of software architecture
- Explain the component relationships in MVC and MVC variants
- Describe and compare different architecture styles including client-server, MVC, layered and tiered architecture.
- Describe and compare SoA, REST and Microservices.
- Select proper architecture styles based on the design goals

- Describe and compare specification inheritance, implementation inheritance and composition.
- Explain basic design principles
- Describe MVC related design patterns: observer, strategy and observer patterns and identify these patterns from given problems.
- Design software architecture of their group project
- Create the skeleton code of their group project and implement a couple of simple functionalities.

Introduction

In the last Module, we discussed various techniques and practices to elicit and analyze requirements. Now we need to translate these application domain requirements to the solution domain design.

Software design is a process to translate the application domain problems (requirements) to the solution domain (software). A “software design” is often used as a representation or model of the software to be built too. It is usually in the form of a set of documents or diagrams, and should be easily translated to code by programmers.

While the requirement analysis focuses on the understanding of the application domain, software design focuses on bridging the application domain to the solution domain. It includes both the high-level design and the detailed design. The output of the high-level design is usually the software architecture. The output of the detailed design is a collection of classes/methods, algorithms, the database schema, design patterns used etc. The detailed design may be blended with the implementation in the Agile software development, reflecting the design directly in the code, instead of separate documents. However, it is always important to have high level design documents to describe the software architecture.

As the cost of hardware rapidly sinking and computer software technology constantly changing with new emerging programming languages and frameworks, the design should be able to cope with all these changes, and the time in which design decisions have to be made need to be kept short.

While a basic architecture may be produced at the beginning of the project, the architecture may need to be refined in each iteration in Agile environments?

Topic 1: High-level Design - Software Architecture

There can be various software design solutions for the same application requirements. It is important to identify possible solutions, evaluate each and choose a proper one. The design goals should be set as the guideline or criteria for this process. As we design our system, we want to select the design that can help achieve our design goals and promote the required quality attributes defined in the requirement analysis phase (usually as the

nonfunctional requirements). We want to prevent the creation of an architecture that is over complex or strives for unnecessary elegance at the expense of critical system properties. As changeability is an inherent feature of software, our design should also reponse the change well.

Design Goals

Before we make any design decision, we need to first set the design goals. First, our design should be sufficient to satisfy the requirements. Second, it should also be flexible to handle changes. In addition, there are also a number of implicit software qualities as introduced in Module 1.

- External Quality Characteristics:
  - Efficiency, often evaluated using the response time and throughput.
  - Reliability: often measured in terms of mean time failure.
  - Robustness: how well a system can cope with errors.
  - Usability: how easily used by the users
  - Security: how the software system is protected from possible malicious attacks
  - Cost: the cost of developing, maintaining and operating the software system. Often measured based on person-hour or person-day spent.
- Internal Quality Characteristics:
  - Maintainability: the degree to which an application is understood, repaired, or enhanced.
  - Re-usability: expected reuse potential of software design or code.
  - Readability: how easily the design can be understood
  - Portability: the ability of software to be transferred from one platform to another.

While all these qualities are desirable, understand that there are always trade-offs. For example,

- Functionality vs. Usability: more functionalities may indicate the decrease of usability
- Cost vs. Robustness: improving robustness will increase cost too.
- Efficiency vs. Portability: Portability to different platforms implies less efficiency usually because lacking of the platform specific optimization
- Cost vs. Reusability: Additional effort is needed to make the code more reusable.
- Backward Compatibility vs. Readability: additional backward compatibility code can make the code hard to read

Therefore, It is important to decide the priority and balance of those qualities in your design goals. Nonfunctional Requirements can usually help define design goals. For example, the top goals of the ProjectPortal project are reliability, usability, portability, and security.

Software Architecture

Software architecture defines the software structures. IEEE defines software architecture as follows:

> *Architecture is the fundamental organization of a software system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.*

Decomposition is a prime method used by software designers to tackle the complexity, an essential difficulty in software. The main task of designing the software architecture is to decompose the software system into a number of smaller subsystems (components) to reduce system complexity while allowing change. As a lot of frameworks exist nowadays to help develop software, the software architecture is also mingled with the technologies used to build the software. Different architectures will address the above -abilities in various degrees. It is important to choose the proper architecture solution that helps accomplish your design goals.

The software system can be decomposed into subsystems (or components) based on the functions, features, data flow, processes, events or objects. Each component usually implements a set of functionalities or provides one or more services to other components. The subsystems can be further decomposed into sub-subsystems. There can be multiple levels of decomposition. The architecture design defines the interface (what functions/ features/data/processes are implemented or services provided by each component) , constraints of components (under what conditions to be used, what characteristics do they have), and the relationships among the components (how do they communicate with each other). The detailed design will define how each component should be implemented.

Cohesion and coupling can help measure the dependency and complexity of the software architecture. Cohesion measures dependency within the subsystem and Coupling measures dependency among subsystems.

- High cohesion: The smaller elements (e.g. classes) in the subsystem perform similar or relevant tasks and are related to each other via many associations.
- Low cohesion: Lots of miscellaneous and auxiliary classes in the subsystem, almost no associations
- High coupling: A lot of connections between subsystems. Changes to one subsystem will have high impact on the other subsystem
- Low coupling: A change in one subsystem does not affect any other subsystem.

In general, we would like to achieve high cohesion and low coupling, so that subsystems do not affect each other and components within the subsystems are related to each other to perform a coherent task. However, there are also trade-offs. To achieve high cohesion, we would like to decompose a system into smaller subsystems, while to achieve low coupling, we want to keep the total number of subsystems small. The hierarchical structure may address this problem. The subsystem can be further decomposed into finer-grain sub-subsystems. Usually the number of top-level subsystems is 3-5.

A principal in decomposition is the separation of concern. Each subsystem (component) should focus on a single concern, and communicate with other components through a well defined interface. The interface should be general enough so that it does not need to be changed constantly.

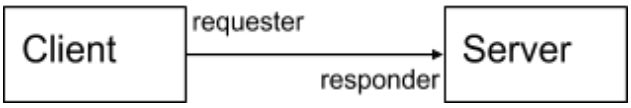
Unified process (or RUP) uses architecture centric methods, putting a lot of emphasis on architecture and software design. While Agile values working software more than comprehensive documents, and does not promote big up front design, architecture design is still a critical task in Agile. Avoiding or delaying architecture design will incur technical debts and degrade software quality. Usually a combination of upfront design and architecture refinement (or refactoring) in each iteration is often adopted in Agile. The amount of upfront architecture design will vary in each project. A good architect will be able to find the right balance between these two for a given situation.

When working with a legacy system, understanding the architecture and structure of the system is also essential. While there is usually a specific role of architect in charge of design, it is important for the whole developer team to understand the architecture. If current architecture cannot accommodate new requirements anymore, possible changes and alternatives should be analyzed and evaluated carefully.

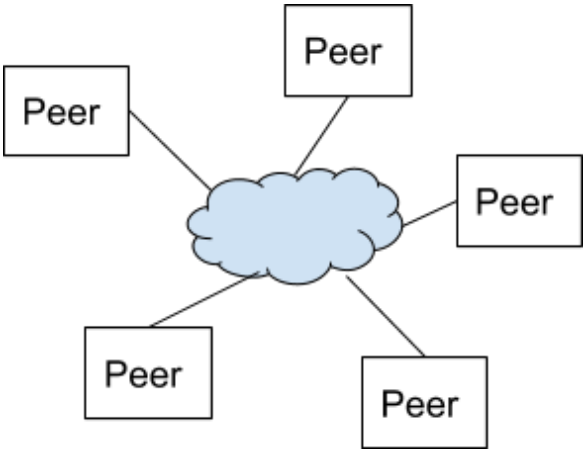
There are several architecture styles that are commonly used either directly or as a basis for different systems. Choose a proper one based on your design goals. A software system can also adopt several possible appropriate architectures in different levels. Let us first introduce several common architectural styles.

Client - Server and Peer-to-Peer

Client-server is a very common and simple architecture style. The system has two different components, namely the client who initiates the service request and the server who responds to the request and provides the service to the client using a request/reply protocol. In this architecture, the relation is asymmetric and a unidirectional relationship. The client needs to know the server interface, but the server does not know anything about the client. A server can serve many clients. Through this separation, clients and the server are often executed in different processes, even on different nodes. A lot of systems use this architecture, such as web server, file server and database.



On the contrary, in the Peer-to-Peer architecture, each peer is both a client and a server. It can initiate requests to other peers as well as response requests from other peers. Without a centralized server, P2P usually provides better scalability. P2P has been used in chat applications such as Skype, and file sharing applications Gnutella and bitTorrent. P2P also supports real-time notification and store-and-forward. However, P2P are harder to maintain as each peer needs to update itself for changes. As peers can come and go, the quality of service may not be predictable. In addition, more security issues can be exposed with vulnerable or malicious peers.



MVC

The Model-View-Controller is an architecture style (or pattern) that decomposes a software system into three main logical components: the model, the view, and the controller.

- Model: application data and logic
- View: presentation of the model.
- Controller: bridge between view and model to carry proper actions.

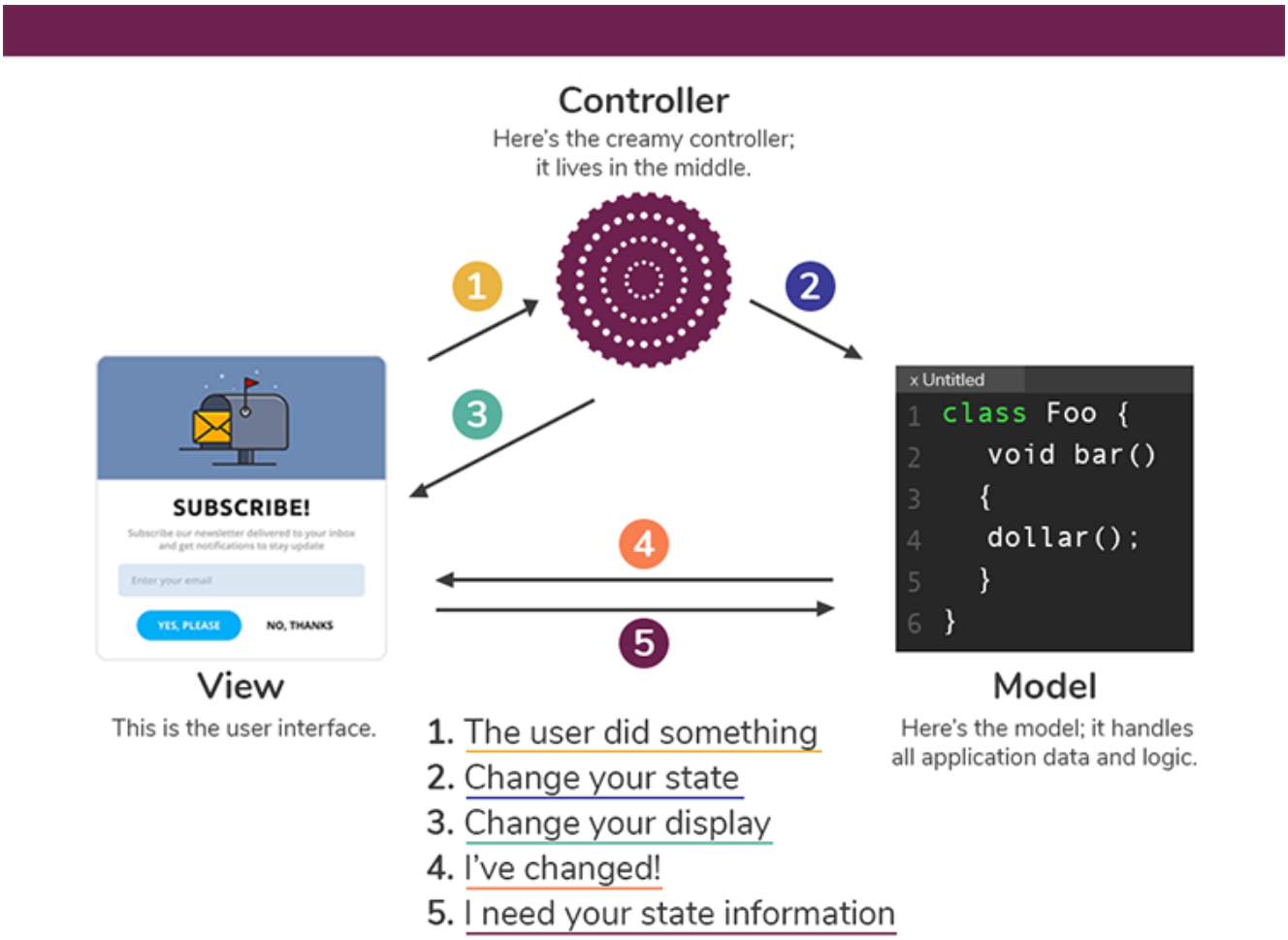
The idea in this architecture pattern is to make the system more resilient to changes by the separation of duties. In general, the presentation changes more frequently than the model. Therefore, It is necessary to separate the application data and logic from the presentation data to the user. The controller mediates between the model and the view, and thus minimizes the dependency between these two. MVC can make the software more reusable and expressive.

In the previous module, we have seen the similar idea used in the Entity-Control-Boundary pattern in the requirement analysis phase. Actually, we can see these two patterns map each other well.

MVC	**ECB**
Model	Entity
Controller	Control
View	Boundary

MVC was first implemented by Trygve Reenskaug in 1979 on Smalltalk at Xerox labs. It was mainly used in GUI applications. Nowadays, it is also a common architecture for web applications. Most industry-standard GUI and web frameworks implement the MVC architecture, Java Swing, Spring (in Java), Ruby on Rail, and Yii (in PhP) are a few examples.

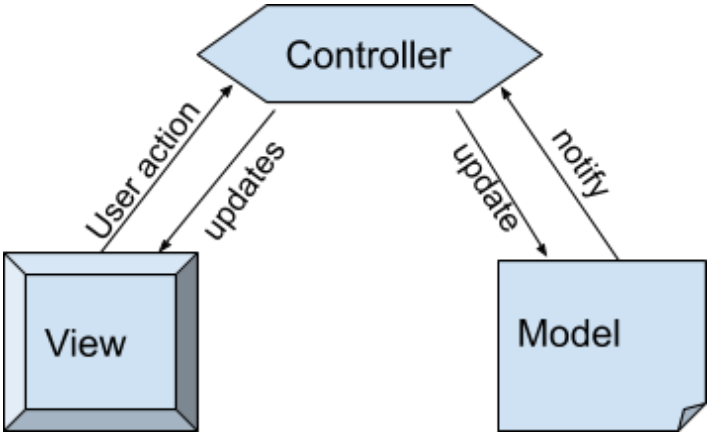
Let us first look at how these three components in the original MVC architecture communicate with each other in GUI applications. The following diagram from the “Head First Design Pattern” book clearly describes the interactions.



- View: Gives you a presentation of teh model. The view usually gets the state and data it needs to display directly from the model.
- Controller: Takes use input and figures out what it means to the model.
- Model: The model holds all the data, state and appication logic. The model is oblivious to the new and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.

MVC Architecture (Based on: Freeman el., 2004)

We can see from the above diagram that the view and the model may still communicate directly without going through the controller. This is the original version of MVC. A modified MVC shown in the following diagram completely eliminates the direct communication between M and V, and makes the C as always the middle man. By completely separating M and V, this version of MVC promotes maintainability and reusability over performance, and has become very popular nowadays.



MVC Architecture

MVC is sometimes viewed as a compound design pattern, which includes three design patterns: composite, observer, and strategy. We will cover these patterns in the next section.

Exercise

Suppose a networked video card game allows a player to interact by means of the mouse and keyboard. A player can view her own deck, the discard pile, or facing cards of the other players at remote sites. The player can also view the pot of money on the table and her own funds. Typically, a player has several views open at once. Explain how you could use the model/view/controller architecture to implement the game. In particular, explain the responsibilities of the model, the views and the controller.

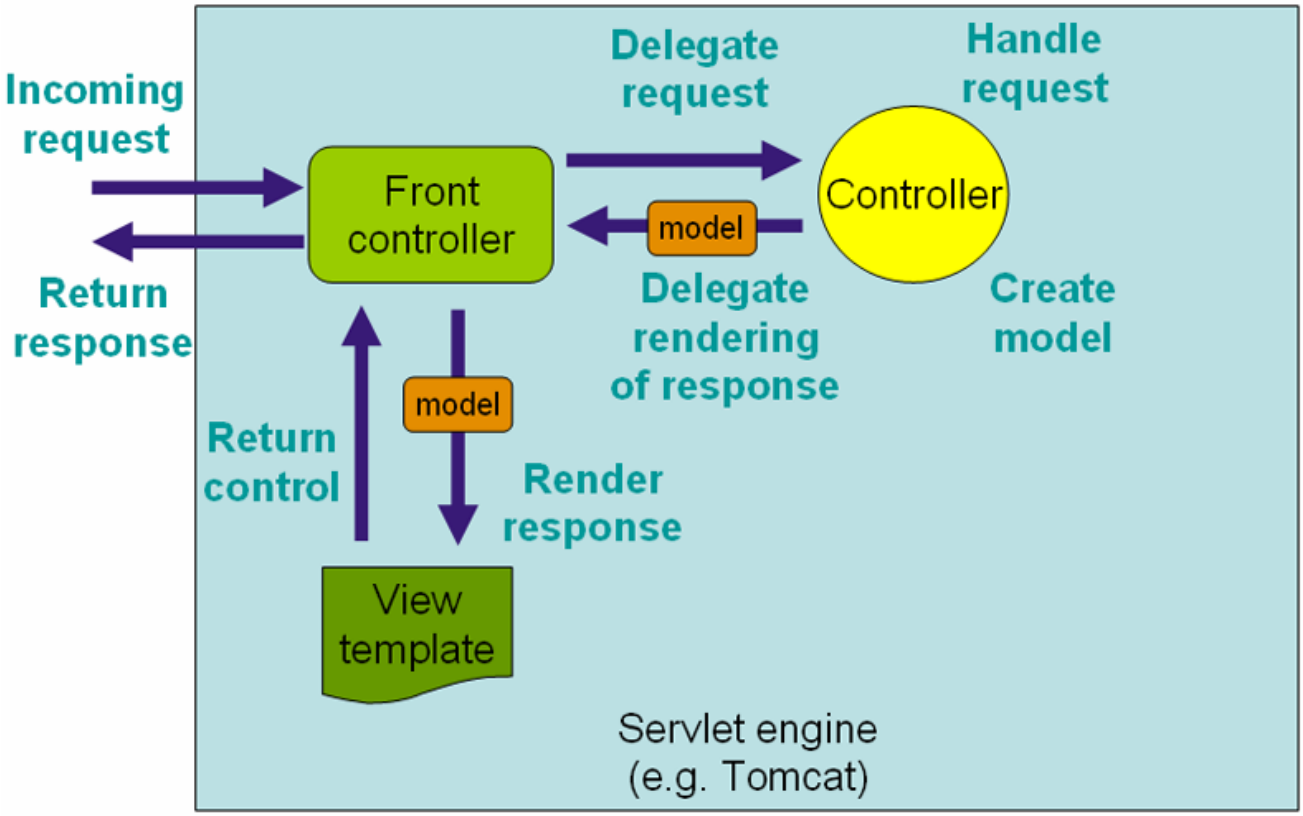
MVC in Web Applications

MVC is also used frequently in web applications. Similarly, the model is responsible for application-domain data and logic. It is often directly related to the database, knowing nothing about the HTML, or the web server. The view displays the model. It can be implemented using templates to render HTML pages. The controller handles incoming requests received from the browser and then passes the proper actions to the model. It can also select an appropriate view to handle the response. It is usually responsible for common actions like authentication and session management.

Decoupling the model and the view ensures that the presentation is always independent of application data. Decoupling the view and the controller ensures that the request made by the user can be independent of the resulting pages.

MVC is implemented by a number of industry-standard web frameworks. Spring web MVC is an MVC framework to develop web applications using Java. The following diagram shows the architecture work flow.

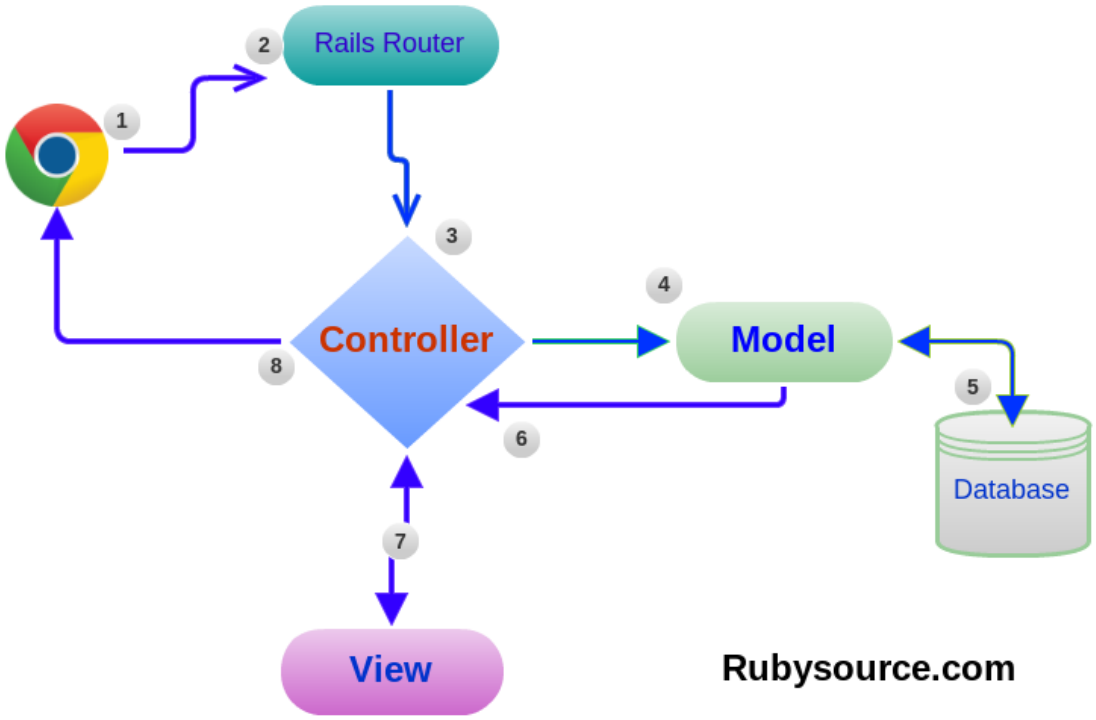
1. An incoming request is first sent to the front controller (DispatcherServlet)
2. The front controller sends the request to a proper controller that will handle it based on the request headers.
3. The controller processes the request, sends it to the suitable service, and then receives the model from the service or data-access layer.
4. The controller sends the model to the front controller (DispatcherServlet).
5. DispatcherServlet finds the view template, using view resolver, and sends the model to it.
6. Using the view template, the model and view page are built and sent back to the front controller.
7. The front controller sends the constructed view page to the browser to render it for the user request.



Source: Web MVC framework

Spring Framework

Ruby on Rail is another famous MVC web framework. The following diagram shows its architecture workflow.

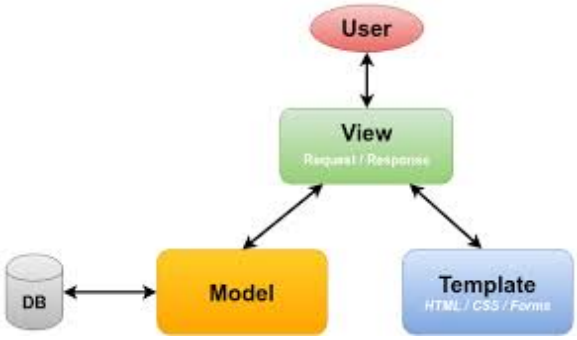


Source:MVC and Rails

Ruby on Rail Framework

1. The browser sends a request to your Rails application.
2. This request gets picked up by the Rails Router (see Rails Routing for more details).
3. The router determines which controller action is responsible for responding to the request and it redirects it accordingly.
4. If data is required from the database, the controller will request it from the model.
5. The model will query the database to obtain the required data.
6. It then returns the data back to the controller.
7. The controller will then pass the data to the view and request the applicable template. The view takes the supplied data and uses it to generate a HTML document which it sends back to the controller.
8. The controller will then return the requested resource back to the browser.

DJango is a popular web framework in Python. It uses MVT architecture as shown below. M (Model) is the data access layer. T (Template) is the presentation layer, defining how the data should be displayed on a web page, V(View) is the logic layer, handling the request and describing which data to present. Some argue that MVT is similar to MVC, where V is mapped to C, and T is mapped to V in MVC.



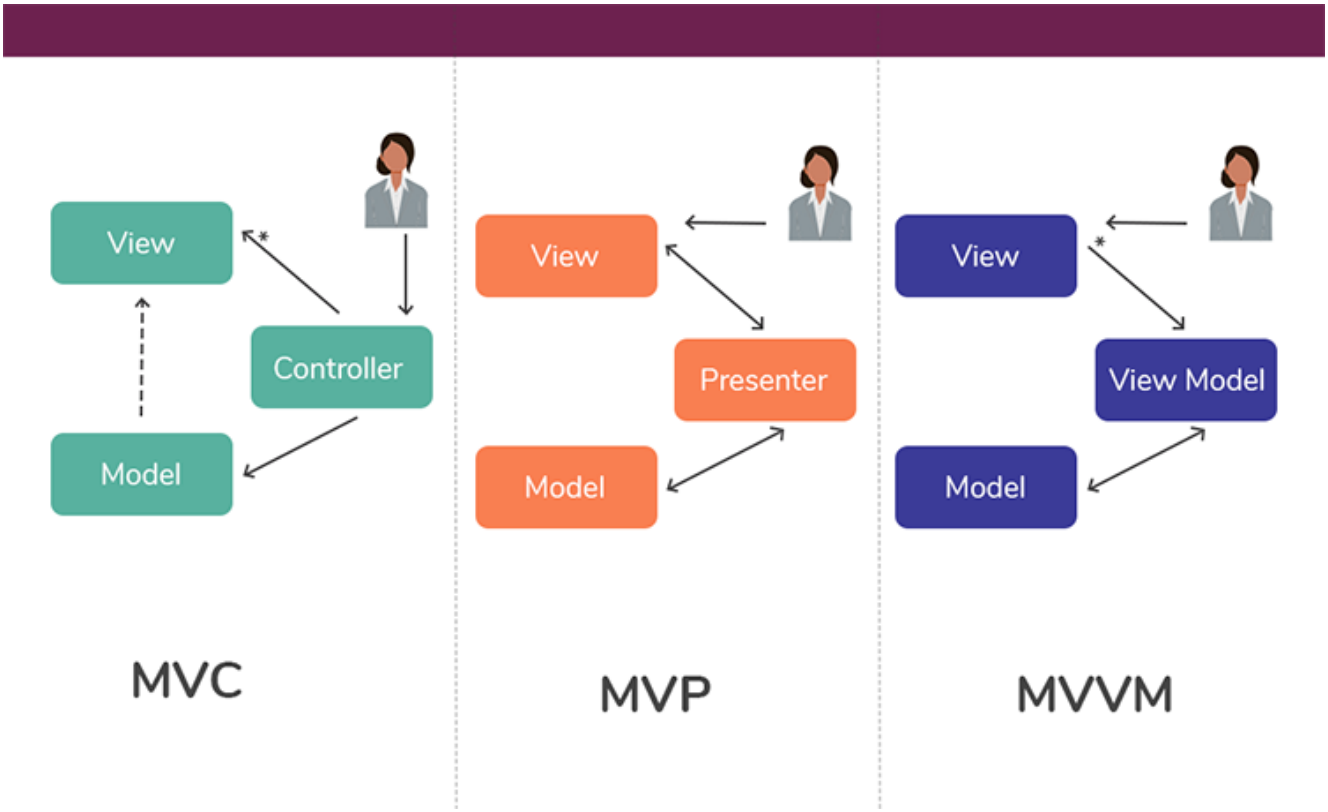
Source: what is Django?

Django Framework

Other frameworks such as ASP .NET Framework, CodeIgniter(php). Angular JS (MVVM) all use some forms of MVC. MVC is also used in Android applications, where Controller is usually the activities, and V is the XML layout files.

As there are some different versions of MVC, sometimes people call all these similar architectures MVWhatever. No matter what functionality the controller should have, all agree that the separation of the model from the presentation is the key. Here are two popular MVC variants.

- MVP: Model View Presenter. Presenter is similar to a controller. It interacts with the model and gets data from the model to update the view. Typically there is a one to one mapping between the presenter and the view, and the view knows the presenter.
- MVVM: Model View ViewModel. A Viewmodel is responsible for wrapping the model and preparing observable data needed by the view. It also provides hooks for the view to pass events to the model. However, the ViewModel is not tied to the view. A ViewModel can be mapped to many views, and the ViewModel has no reference to the view.



MVC vs MVP vs. MVVM

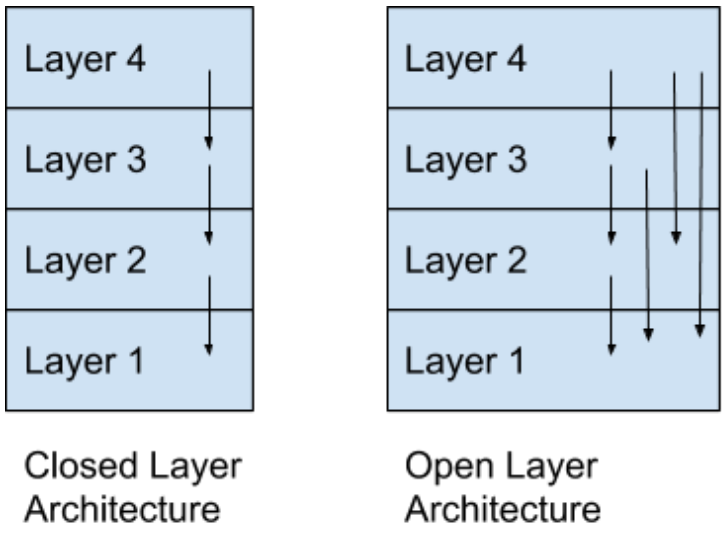
Layered Architecture and Tiered Architecture

In the layered architecture, the system is divided into layers, with each layer providing a related set of services to upper layers. Each layer has no knowledge of upper layers and only depends on services provided by lower layers. The layered architecture decomposes the system not based on features or functions, but based on data and control flow. It restricts communication between layers to a unidirectional relationship.

There are two types of layered architectures:

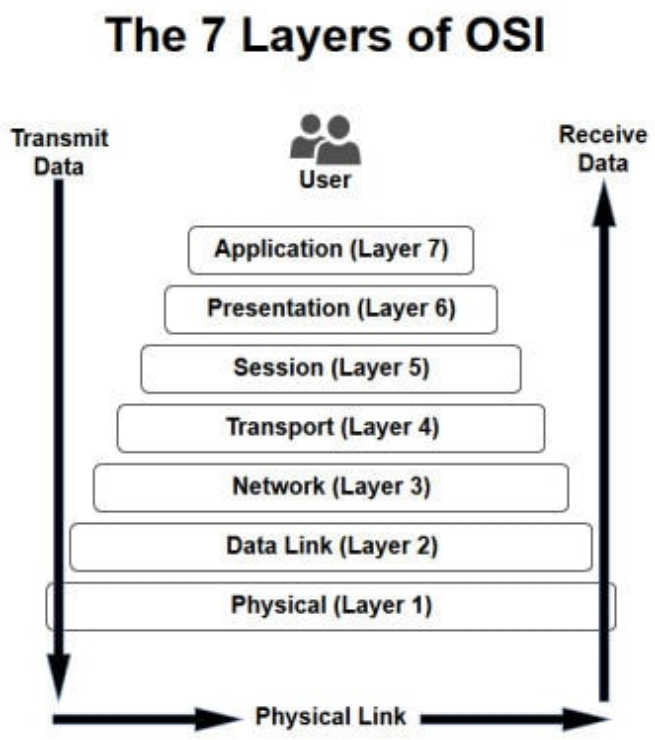
- Closed Layered Architecture (strict): each layer can only call operations from the adjacent lower layer.
- Open Layered Architecture (relaxed): each layer can call operations from any layer below.

The pros of using the closed layered architecture is to minimize the interactions between layers to achieve better flexibility and maintainability. When a layer is changed, it only affects its adjacent upper layer. On the contrary, it is less efficient than the open one, as it lengthens the call path when non-adjacent layers want to communicate. For example, Layer 4 cannot communicate directly with Layer 1. Rather, it needs to call Layer 3 first, then Layer 3 calls Layer 2 and finally Layer 2 calls Layer 1.

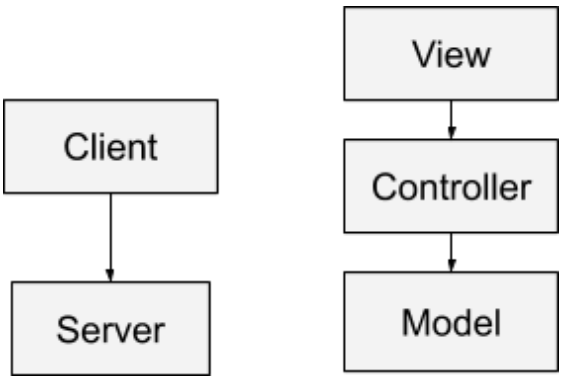


Layered Architecture

A famous example of the closed layered architecture is the OSI 7-layer model as shown below.



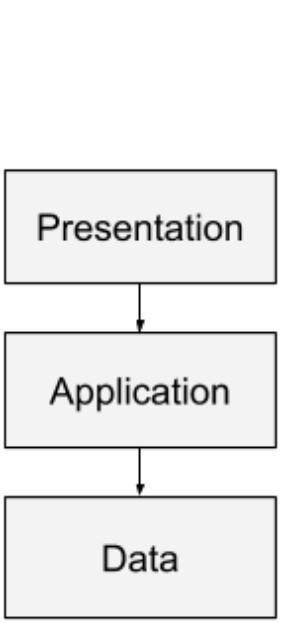
Layered architecture is a generic architecture that defines the relationships between components, not the roles of each layer. We can consider the client-server and MVC (one of the variants) as a special type of the Layered architecture too.



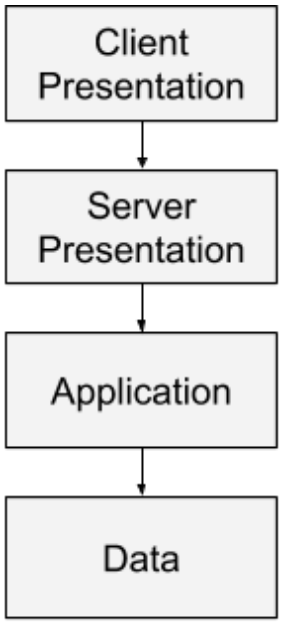
Client and Server cs. MVC

When layers are distributed on physically separated nodes, it is called Tiered Architecture. As the client and server are usually on different nodes, we can consider it as 2-tiered architecture. More commonly are 3 -tiered or 4-tiered architecture.

In a popular 3-tiered architecture, the bottom tier is the Data tier, usually in the form of a database, maybe a separate database server such as MYSQL, PostgreSQL, Oracle, Mongo, etc. The middle tier is the Application tier to process the business logic for the application. This can be an application server written in C#, Java, C++, Python, Ruby, etc. The upper tier is the Presentation tier, representing the GUI or web pages in the form of HTM/JS/CSS.



A 3-tiered Architecture



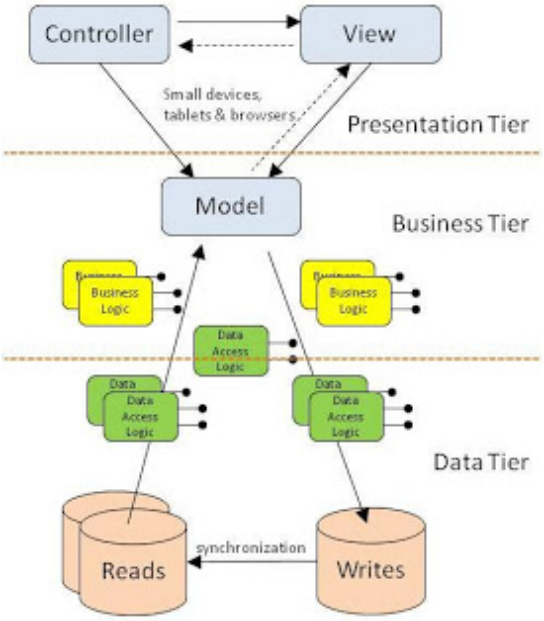
A 4-tiered Architecture

Tiered Architecture

Sometimes, we can even split the Presentation tier into two separate components: client presentation and server presentation, which is a 4-tiered Architecture.

The tiered architecture provides the same benefits as the layered architecture, maintainability, reusability, flexibility, etc. By separating them into different nodes, it also enhances the security and performance.

In many cases, multiple architecture styles may be used together in a software system. For example, MVC can be used together with the tiered architecture as shown in the following diagram.



Source: MVC in a three-tiered-architecture

Service Oriented Architecture

Service Oriented Architecture (SOA) is an architecture style where an application's business logic or individual functions are modularized and presented as services for consumer/client applications. Targeting large applications at the enterprise level, SoA provides business automation and organizational agility through service abstraction and loose coupling between business and application logic.

In SoA, each service is a self contained, reusable unit of software that executes a certain task independent of the technology used. Each service can be implemented using different architectures, programming languages and implementation stacks. Services can communicate with each other, and possibly be replaced or combined with other services.

The service interface is independent of the implementation. The service consumer can use a totally different technology stack from the service provider. For example, a service can be implemented in Java, and the consumer can use .NET. They are loosely coupled.

Application developers or system integrators can build applications by composing one or more services without knowing the services' underlying implementations. For example, Instead of building each component from

scratch, we can leverage existing services such as the facebook authentication service, paypal gateway payment service, and Google map service and quickly develop a new application.

The services are exposed usually using standard network protocols—such as SOAP (simple object access protocol)/HTTP or JSON/HTTP—to send requests to read or change data. The services are published in a way that enables developers to quickly find them and reuse them to assemble new applications. However, be aware that SoA does not require web services though they are commonly coupled with SoA.

Similar to other architecture styles, SoA is designed based on the basic Separation of Duties principle, and promotes reusability. Through reusing services for different purposes, productivity is increased, particularly at the organizational level. Unlike architecture styles discussed previously, services are much more loosely coupled with greater flexibility, autonomy and interoperability. This enables the new application systems to work together with existing systems and legacy applications. Implementing SoA can be costly. It may outweigh the benefits for some organizations. So each case should be analyzed separately to evaluate if and how to adopt SoA.

Thomas Erl's discussed the following SoA features in his book "Service-Oriented Architecture".

- Standardized service contract: Each service should have a standardized service contract, consisting of a technical interface and service description.
- Service loose coupling: Services should be loosely coupled and independent of each other. Service contracts should be designed to have independence from service consumers and from their implementations.
- Service abstraction: Service contracts should only contain information that it is necessary to reveal, and detailed service implementations should be abstracted out.
- Service reusability: Services should be designed with reusability in mind, with their service logic being independent of a particular technology or business process.
- Service autonomy: Services should be designed to be autonomous, with more independence from their runtime environments.
- Service statelessness:State data should be separated from services.
- Service discoverability: Services should be discoverable by humans who are searching manually as well as software applications searching programmatically. Services must be aware of each other for them to interact.
- Service composability: Services should be composable. They can be combined with other services to create new services or applications.

Microservices

Microservices, aka microservice architecture, is another service centered architecture style. It structures an application as a collection of small autonomous services modeled around a business domain. The main idea behind a microservice architecture is also divide and conquer, not only at the design and development phases,

but also at deployment and operation phases. As the modern applications get more and more complicated, it is easier to build and maintain when they are broken down into smaller pieces that work seamlessly together. The microservices go beyond the modular design. Each service is not just a component that can be independently designed and developed, but also can be independently implemented using different technology stack, independently deployed in different environments and individually scaled.

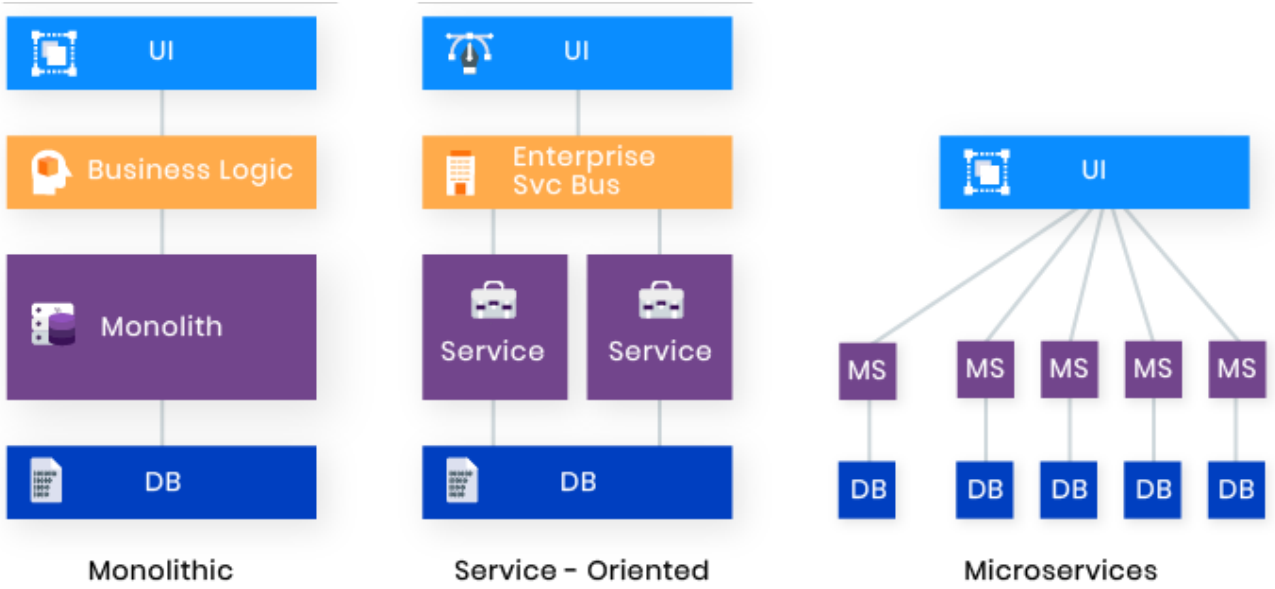
A microservice architecture is particularly well-suited for large and/or complex software systems. With the Microservice architecture, the whole application is not a monolithic that needs to be re-installed when changes are made. The whole application is loosely coupled, with only the need to replace or re-install certain services when changes are made. Therefore, decentralized continuous delivery is made possible and necessary to effectively develop, maintain and operate such applications.

Unlike the service components in SOA that may range in size anywhere from small application services to very large enterprise services, microservices are generally single-purpose small services that only do one thing well. In addition, each microservices has its own private data store while services in SoA may share data. Though services in both SoA and Microservices are very loosely coupled, and can be implemented using different programming languages, such as technology stacks and environments, microservices can be individually scaled and independently deployable.

Microservices communicate with each other using well-known, lightweight message protocols. Though no rule or restriction to what protocol should be used, a common implementation for microservices is using HTTP through REST API calls, particularly for synchronous communication. It is common for REST to be used with JavaScript Object Notation (JSON). Service operations can accept and return data in the JSON format as it is a popular and lightweight data-interchange format.

An important design goal of SOA is reusability, Services can be reusable, discoverable, and replaceable. However, a more important design goal of Microservices is separation to achieve deployment flexibility, technology flexibility, and scalability. The separation is around the business domain, not just some reusable technology.

Using a microservice architecture, the overall application has added complexity. It is also not trivial to decompose a complex system into a right set of microservices with proper granularity. Too fine-grained will result in a large number of services, and too coarse-grained will increase the complexity of individual services. In addition, it is also challenging to maintain the consistency of different data store in microservices.



Source: [Best Architecture for an MVP: Monolith, SOA, Microservices, or Serverless?]

Monolithic vs SOA vs Microservices

REST

REST stands for REpresentational State Transfer. It is a software architecture style for Network based applications defined by Roy Fielding in his dissertation in 2000. It is defined as a set of constraints for the components in the architecture:

- Client-server: REST is based on the client-server architecture.
- Stateless (Server does not maintain state): All states should be either turned into resource states, or kept on the client.
- Cacheable: resources must declare themselves cacheable. Caching shall be applied to resources when applicable
- Layered: allow to use a layered system architecture to deploy APIs
- Uniform interface: RESTful HTTP Applications give every resource a URI and link them together through hyperlinks.
- Code on demand (optional): the executable code can also be rendered.

A system conforming to REST constraints is a RESTful system. It is typically used with HTTP and web services. Restful applications use HTTP to perform all CRUD operations. They use standard HTTP operations (GET, POST, PUT, DELETE).

REST is often used together with microservices now, as a way to implement microservices. The stateless HTTP feature of REST is a great way to expose microservices while keeping them decoupled.

Besides microservices, there are several emerging architecture styles for model applications. One is the serverless architecture, with which you do not need to deal with physical servers, and the complexity of how compute resources are provided is hidden from you. Mostly it uses cloud based services, such as FaaS(Function as a Service) and BaaS (Backend as a Service). Cloud-native applications have been designed and developed from the ground up to be deployed in the cloud. In doing so, applications can take full advantage of their deployment environment.

Test Yourself

We should try to minimize both coupling and cohesion in the software architecture design.

True

False

Test Yourself

High reliability means that the software can still perform well under error.

True

False

Test Yourself

Nonfunctional requirements should be used to set the design goals.

True

False

Test Yourself

The Client-server architecture is more scalable than P2P in general.

True

False

Test Yourself

The data should be separated from its presentation.

True

False

Test Yourself

Describe in your own words the similarities and differences between original MVC, MVP and MVVM.

Test Yourself

In both SoA and microservice architecture, services can be implemented using different technology.

True

False

Test Yourself

Describe in your own words the similarities and differences between SoA and Microservices.

Test Yourself

Stateless is an important feature of REST.

True

False

Topic 2: Design Principles and Design Patterns

Following the architecture design, detailed design focuses on the design of each component or subsystem. Object oriented design is usually used nowadays in the application software development to specify classes, attributes, methods and their relationships. Class diagrams are the outcome of this activity. Detailed design also includes database design, algorithm design, as well as applying design patterns. It addresses major design goals such as flexibility, reusability, maintainability, and reliability.

Class Diagrams

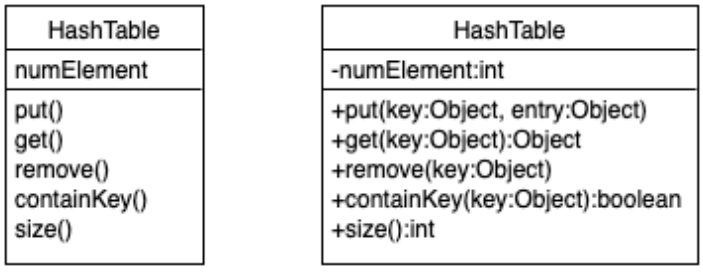
In the previous module, we have discussed class diagrams that can be used in the requirement analysis phase, in which domain classes and their relationships are identified. In the design phase, additional classes are identified from the architecture, frameworks, or design patterns to address the design goals. Moreover, the classes are described in more details, including

- Visibility: classical object-oriented languages, such as C++ and Java, use the visibility modifiers to control the access to class resources. The idea behind it is to ensure the principle of data encapsulation. There are usually three modifiers: public, private and protected. Only public members can be accessed from outside of the class. Private members can only be accessed from inside. Protected members can be accessed from the class as well as the subclasses. The following table shows the visibility of the class members accessed by different types of developers.

Developers	Tasks	Visibility
Class Users	Call class	Public Members
Class Extenders	Extend Class	Public and Protected Members
Class Implementers	Implement Class	Public, Protected, and Private Members

- Member fields and their types: not only do we need to specify the attributes of the class, but also need to specify their types in the design phase.
- Methods and signatures: not only do we need to specify the methods of the class, but also need to specify their signatures in the design phase. The signature of a method includes number of parameters, parameter types, the parameter order and the return type.

As shown in the following diagram, the left side is acceptable in the requirement analysis phase, additional details such as visibility, types and method signatures are added into the class diagram in the design phase as shown in the diagram on the right side.



- Constraints: Boolean conditions can be specified to member fields or methods in the class diagram to represent some condition, restriction or assertion related to them. They are called constraints, and must be

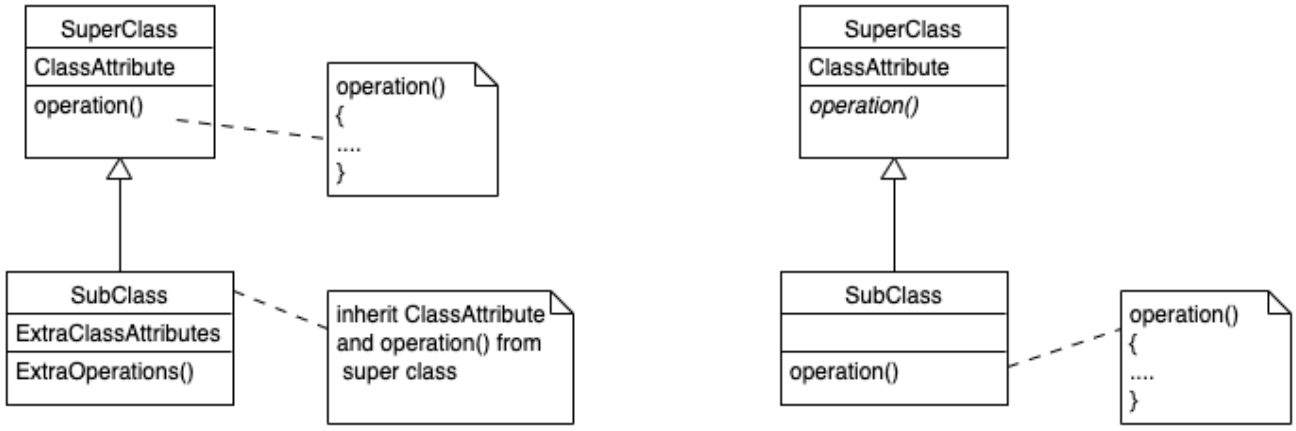
satisfied (i.e. evaluated to true) by a correct design of the system. There are three types of constraints: invariants to apply on member fields, precondition and postcondition to apply on member methods. We will discuss this in more detail in later modules.

- Relationships: the following class relationships are discussed in the previous module.
  - Inheritance
  - Aggregation
  - Composition
  - Association
  - Dependency

OO Ways of Reuse

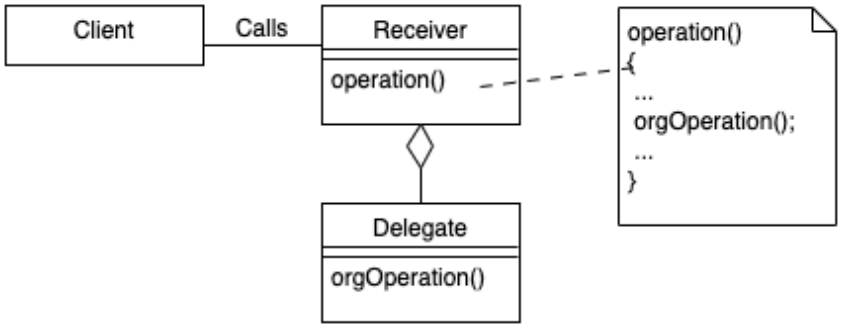
Reusability is very important in software design. It can result in increased productivity and quality by minimizing the risk of newly developed projects. There are two ways that we can reuse current code in OOD (Object Oriented Design): inheritance and composition. Through inheritance, new subclasses reuse all public or protected members in the superclass, and additional new functionalities are added too. In this case, access to the superclass must be available. Through composition, new classes can reuse the implementation of certain public methods without knowing other details. New functionalities can be implemented by wrapping the existing functions. In this case, the access to the original class is not needed.

Inheritance can support both implementation and specification reuse. Specification inheritance. With implementation inheritance, **both interface and implementation of methods** in the superclass are inherited by subclasses. The goal is to reuse functionalities already implemented in the existing class. With specification inheritance, **only the interface** is inherited. The goal is to reuse specification, which is usually an abstract class or interface with all operations specified, but not yet implemented. Polymorphism is inherently related to the specification inheritance. The following diagrams show the difference between these two types of inheritance.



Implementation Inheritance and Specification Inheritance

The following diagram shows how to reuse existing code (in Delegate) to implement new functionality (in Receiver) using composition. By composite Delegate into Receiver, the Receiver can delegate the client request to the Delegate. For example, if a Client wants to call an operation(), instead implementing this function from scratch, we can reuse the implementation code in the orgOperation() in the Delegate to implement the operation() method.

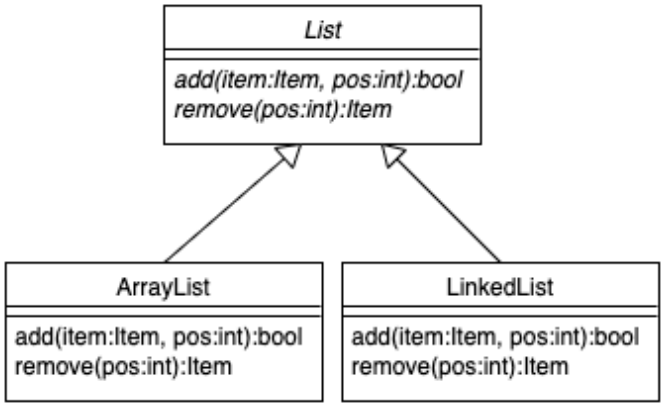


Composition/Delegation

Examples

1. I need a List class that can add or remove an element from the list. It may be implemented as an arraylist or a linked list.

   The requirement here is to have the same interface (add() and remove()), but with possibly different implementation. We should use specification inheritance, as shown in the diagram below.

2. Now I have the List class with operations such as add() and remove() that is implemented either using ArrayList or LinkList. Later I also want a Stack class which should have operations such as push(), pop(), top()). Should we use inheritance or composition to reuse the List class?

   The requirement here is to reuse the existing code, but have different specifications. We want push() and pop() instead of add() and remove().

It is better to use composition than inheritance. If using inheritance, Stack will have undesired behaviors, such as add or remove an item into or from any position.

```
Sample Code (in Java)

class Stack
{
    List myList = new ArrayList();

    Item pop()
    {
        myList.remove(0);
    }

    void push(Item item)
    {
        myList.add(item, 0);
    }
}
```

3. The following table compares inheritance and Composition (Sometimes called Delegation) in code reuse.

Inheritance	**Composition (Delegation)**
◦ Supported by many programming languages ◦ Support implementation reuse	
◦ More coupled. ◦ Exposes a subclass to details of its superclass ◦ Change in the parent class requires recompilation of the subclass. ◦ More efficient. ◦ Specification inheritance is more used to reuse interfaces.	◦ Less coupled ◦ No need to access the details of the delegated class. ◦ It is easy to replace the delegated class to a different one ◦ Less efficient. ◦ Implementation reuse

Design Principles

Reusability is also related to other “-bilities” in the developers’ wish list, such as the following:

- **Flexibility**—Today's system may not meet the requirements of tomorrow's. Our design should be able to cope with changes.
- **Extensibility**—The design should allow the delivery of more functionality after an initial delivery.
- **Maintainability**—The design should minimize the introduction of new problems when fixing old ones.

In general, specification inheritance provides better flexibility and extensibility. New subclasses with different implementations can be easily added without changing existing code, which increases maintainability. Implementation reuse through composition can provide better flexibility and extensibility than implementation inheritance, since the new code is less coupled with existing, reused code.

Here are some well-known principles restating the above guidelines:

- Program to an interface, not an implementation.
- Apply the dependency-inversion principle: Depend on abstractions.
- Favor composition over inheritance for implementation reuse.
- Apply the Liskov substitution principle: Subclasses should be substitutable for their base classes. If an object of type *S* can be substituted in all the places where an object of type *T* is expected, then *S* is a subtype of *T*.

Additionally, the following principles should be used to provide better flexibility, extensibility, and maintainability:

- **DRY (don't repeat yourself)**—Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. A modification of any single element of a system does not require a change in other logically unrelated elements.
- **Single-responsibility principle**—A class should have a single responsibility and only one reason to change.
- **Separate variants from invariants**—Invariants can always be reused, and variants can always be replaced.
- **Open-closed principle**—Classes should be open for extension but closed for modification.
- **Limit the number of classes to interact**—The effects of changes can be limited.
- **Interface-segregation principle**—Split a general-purpose interface into multiple independent ones. This can reduce the side effects and the frequency of required changes.

Some use the acronym "SOLID" for the five most popular principles mentioned above:

- Single responsibility principle
- Open–closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Design Patterns

Applying the above design principles, design patterns are high level ideas on how to structure classes and objects to solve certain problems. Be aware that we need to write our own code to adopt those patterns, instead of using them directly. Framework and libraries are not design patterns, but are specific implementations that can be used in our code. However, they usually make use of design patterns in their implementations.

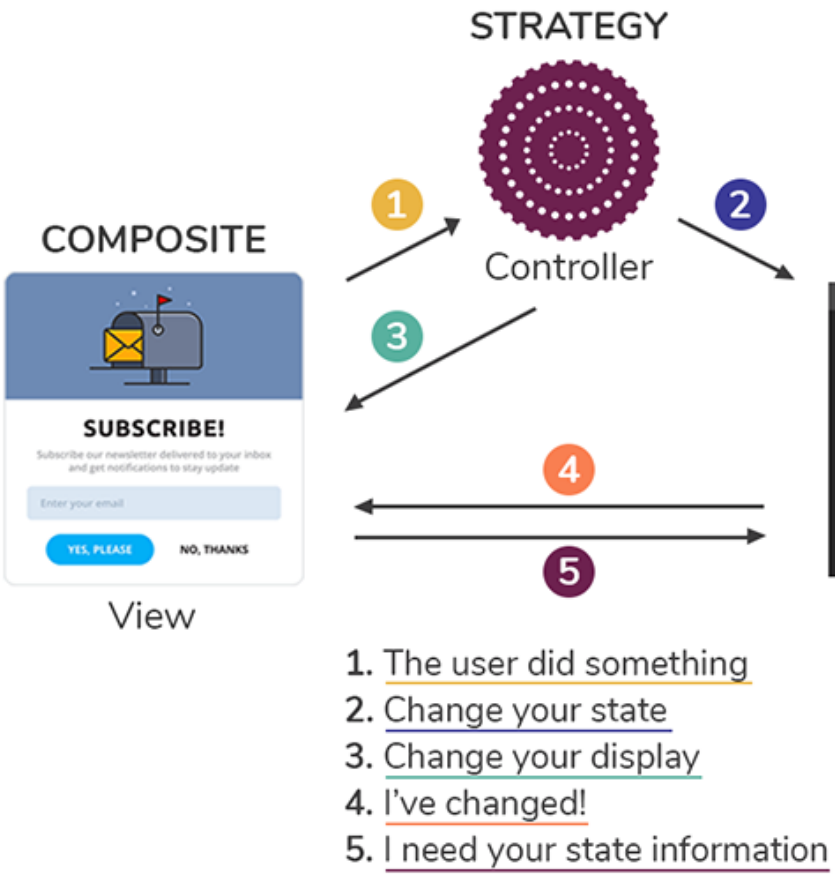
Design patterns provide a shared vocabulary to solve common problems. The famous design patterns book "Design Patterns: Elements of Reusable Object-Oriented Software" was published in 1994 by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, who are frequently called "Gang of Four". This book introduces 23 design patterns in three categories:

- Creational patterns: design patterns about class or object creation, including abstract factory, builder, factory method, prototype, singleton, etc.
- Structural patterns: design patterns about class and object composition, including adapter, bridge, composite, decorator, facade, proxy, etc.
- Behavior patterns: design patterns about class or object communication, including command, interpreter, observer, state, strategy, template method, etc.

Besides the original design patterns book by GoF, "Head First Design Patterns" by Eric Freeman et al. is a fun book to read. It explains some of these design patterns using a visually intensive, engaging and conversational style, and thus makes them easier to understand. The SourceMaking website also provides good free content on design patterns. If you want to learn design patterns in more detail, you can take our design pattern course MET CS 665.

In this module, we will focus on three patterns that are related to MVC: Observer, Strategy and Composite. MVC sometimes is also called a compound design pattern of these three. Through these three design patterns, we hope that you will have a better understanding of the above design principles as well as gain more insights on how to make better design.

The following diagram from the "Head First Design Patterns" (Freemans et al., 2004) book describes how these three patterns are used in the MVC architecture.



The view and controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Usin the Strategy Pattern also keeps the view decopuled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows othing about how this gets done.

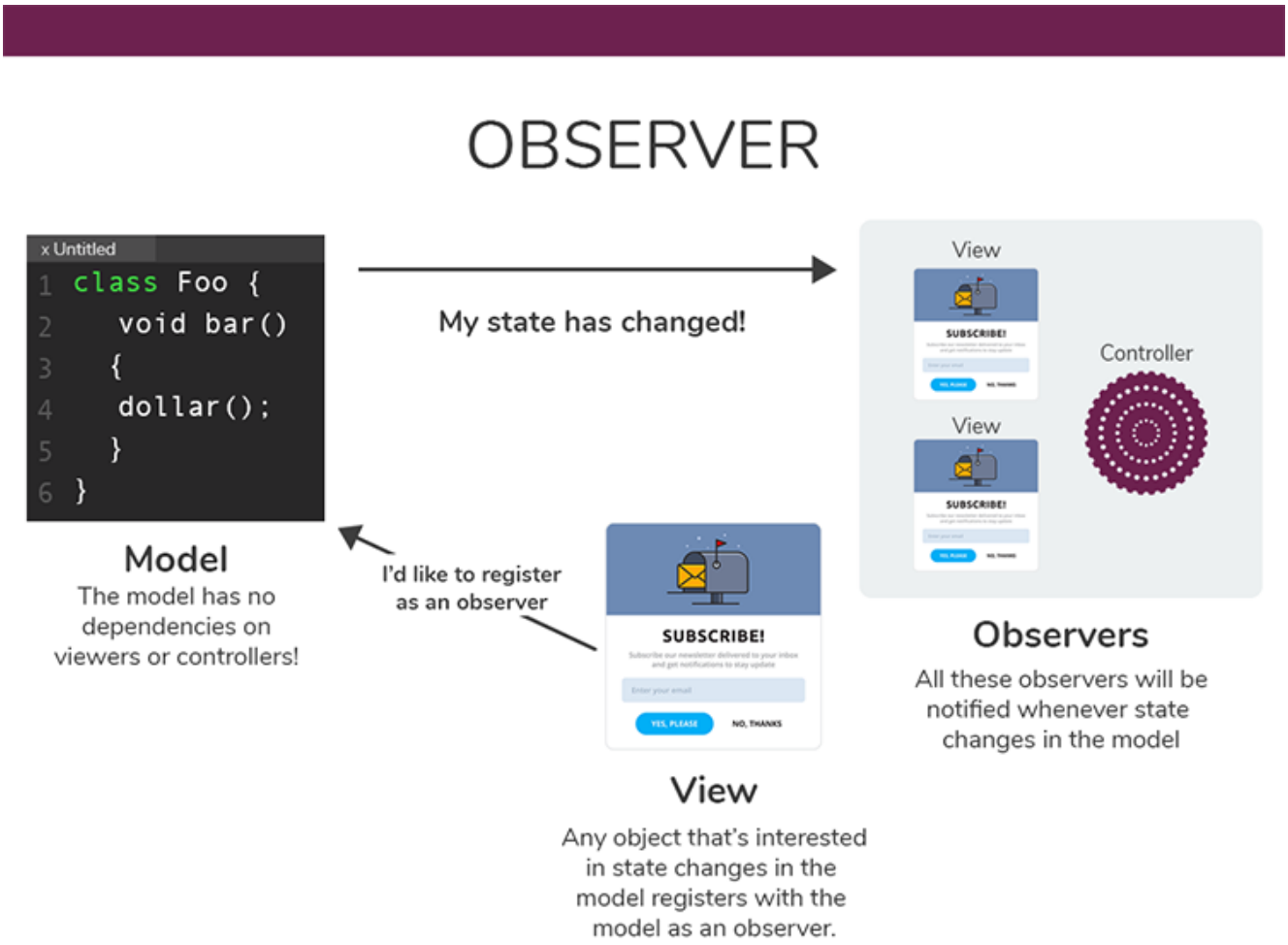
- Composite: The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.
- Observer: The model implements the Observer Pattern to keep interested objectes updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allos us to use different views with the same model, or even use multiple views at once.

MVC Compound Pattern (Based on: Freeman, E et al., 2004)

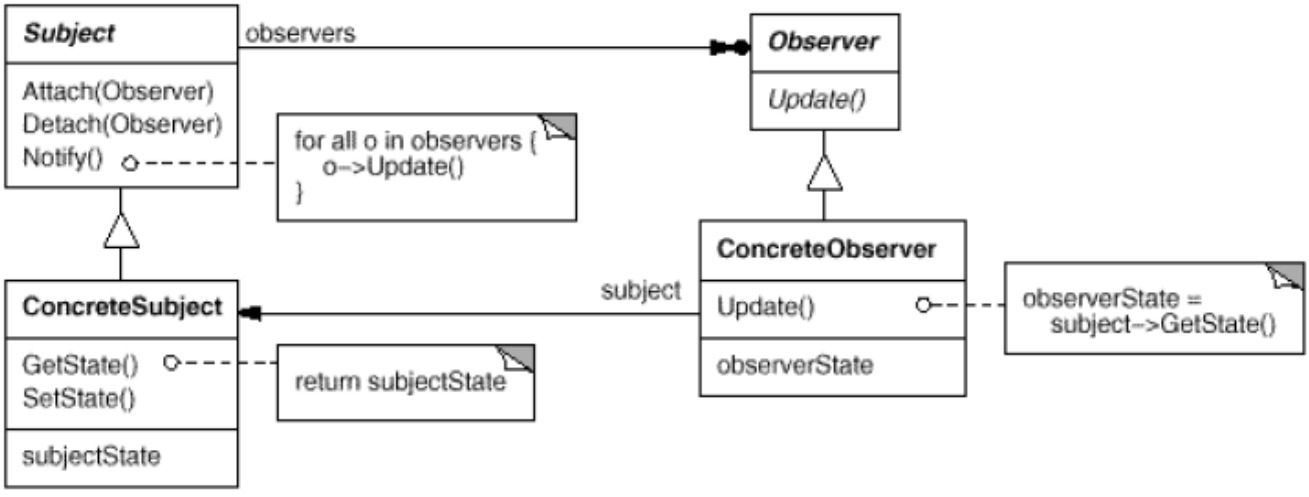
Observer

Suppose we have an object that changes its state quite often, and we need to provide multiple views of the current state of the object, the observer pattern can be used to maintain consistency across all views, wherever the state of the observed object changes. This pattern also makes it possible to add new views easily without

having to recompile the observed object or the existing views. For example, we have multiple views to show one's stock portfolio such as histogram view, pie chart view, timeline view, etc. Since the stock state changes constantly, all views should be updated frequently to reflect the changes. In the MVC architecture, models are observable, and views (or views and controllers together) are observers. Each view is registered to observe some model. Whenever the model changes, all views will be updated.



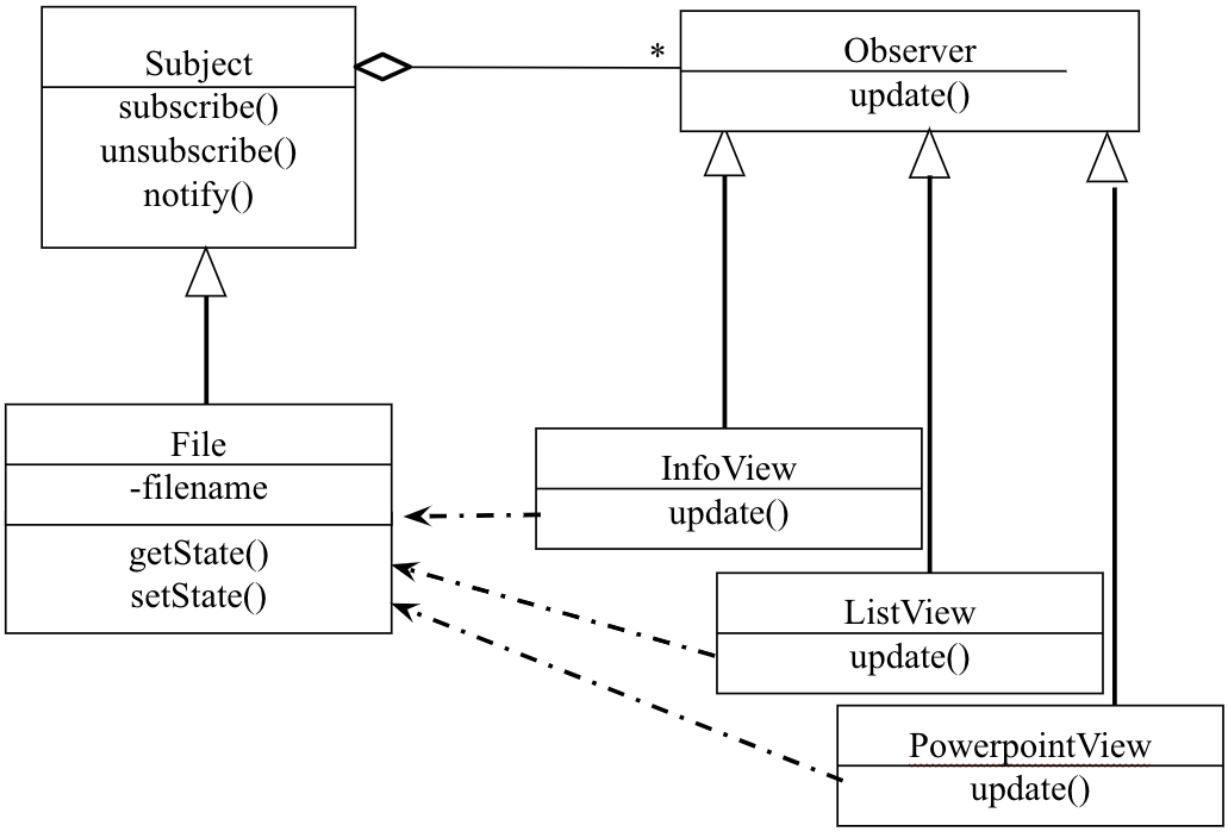
The class diagram of the observer pattern is shown as below:



Observer Pattern Class Diagram (Gamma, E et al., 1994)

The subject should be aware of all observers in order to notify them whenever there are state changes. However, it is hard to know what kinds of views when designing the model (or the subject). Therefore, it is important to apply the principle “Program to an interface, not an implementation” and Dependency inversion Principle. Instead of using concrete Subjects and Observers directly, an abstract Subject is used together with the abstract Observer. All subjects provide three operations: attach() (or subscribe()) an observer, detach() (or unsubscribe()) an observer and notify() all observers upon changes, that will call observers’ update() method consequently. The abstract class (or interface) is general enough to handle any types of subjects and observers. No concrete information is needed in advance. Then in each application, the ConcreteObserver implements the update() method by pulling the information from the ConcreteSubject that it displays. This pattern programs to the interface and decouples the observable and observer, which makes the code more reusable and easily add new observers or subjects without changing the existing code.

Here shows an example to use the Observer pattern in the file system design. Suppose we need multiple views of the file system, when the filename is changed in one view, all other views should be updated too.

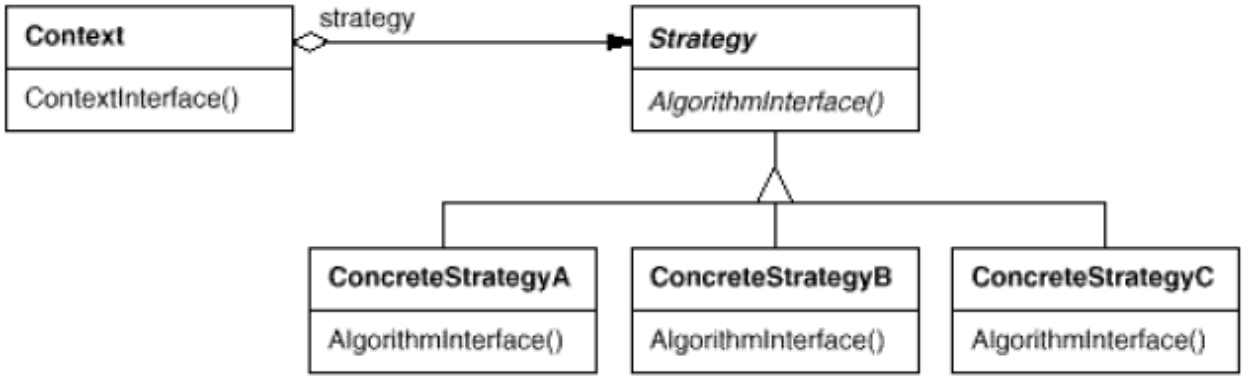


File System Class Diagram using Observer Pattern

Observer pattern is a widely used behavioral pattern. It focuses on how subjects and observers communicate with each other. It defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This pattern is implemented in many libraries and frameworks. Java implements the observer pattern in its built-in Observer and Observable classes (java.util.Observer and java.util.Observable). Most client side JavaScript programs also implement the observer pattern in the form of Ajax event handlers.

Strategy Pattern

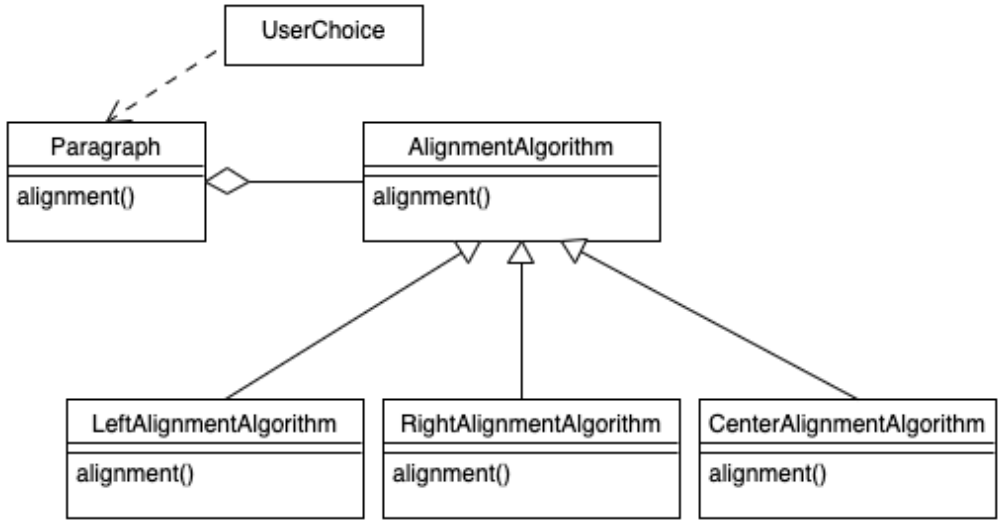
Suppose there are different algorithms that exist for a specific task, we want to switch between these algorithms at run time. Strategy pattern solves this problem by decoupling the algorithm from the contexts that use it. Strategy is another behavior pattern. The main idea is to capture the algorithm abstraction in an interface, and bury implementation details in the derived concrete algorithm classes. Specification inheritance is used again to support the changeability and composition is used between the decoupled context and algorithms. The class diagram is shown below:



Strategy Pattern Class Diagram (Gamma, E et al., 1994)

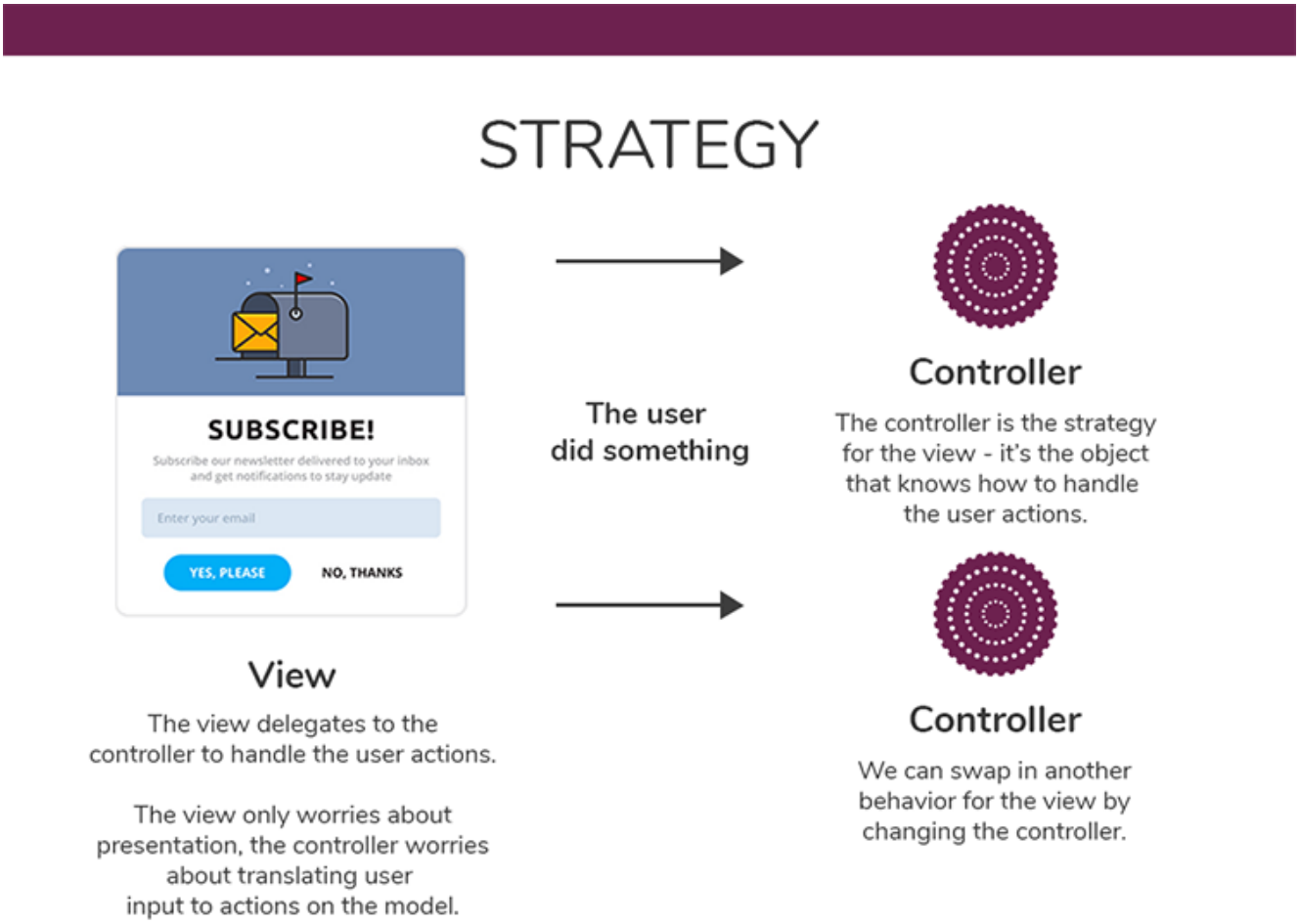
In this diagram, we can see whenever ContextInterface() is called, it is delegated to the AlgorithmInterface() of Strategy, and further through polymorphism, to the corresponding concrete algorithm from a proper subclass, such as ConcreteStrategyA or ConcreteStrategyB or ConcreteStrategyC. This enables the dynamic change of the algorithm from one to another, as well as easily adding new algorithms. Which ConcreteStrategy is best usually depends on an external policy in the current Context.

Here is a concrete example. Suppose we want to implement a paragraph that can use different alignment algorithms. It can be left aligned, or right aligned, or center aligned, depending on the user choice in the run time. We can use the strategy pattern to implement it as below:



Paragraph Class Diagram using Strategy Pattern

Strategy Pattern is used in the MVC when we would like to dynamically change the controller.



(Based on: Freeman, E et al., 2004)

Here are more examples of tasks that can use the strategy pattern:

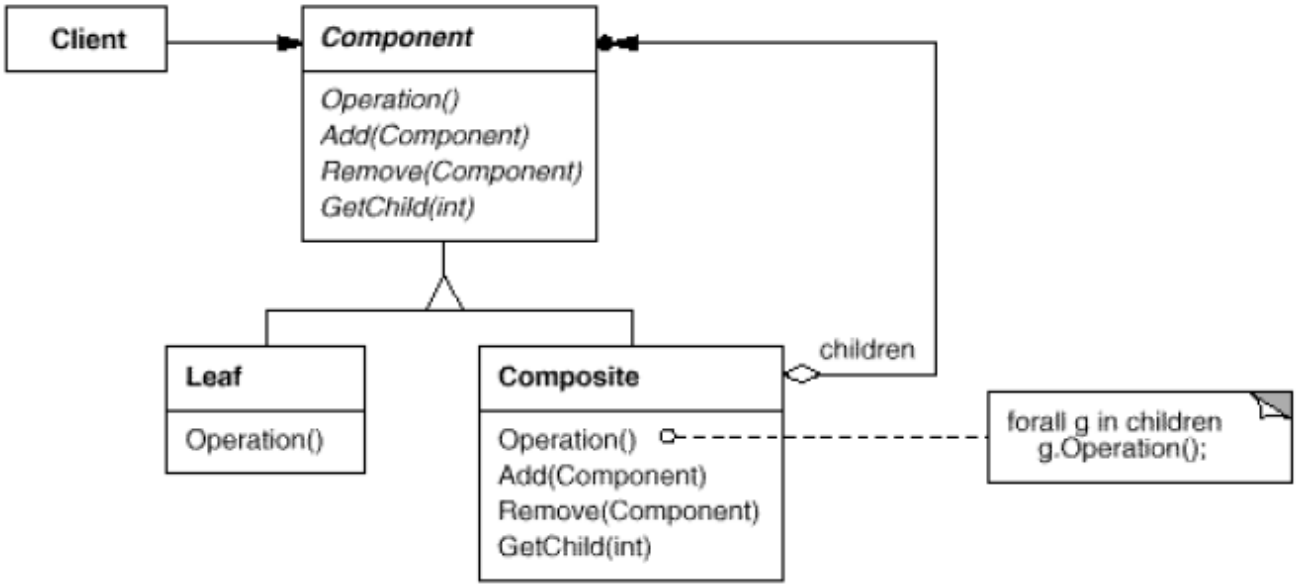
- Implement different collision strategies for objects in video games
- Parse a set of tokens into an abstract syntax tree using different methods such as bottom up or top down.
- Sort a list of data using different algorithms based on current resources, such as Bubble sort, mergesort, and quicksort.
- Implement different actions (such as get, put, post, etc) based on http requests.

Composite

Composite is a structural pattern. It focuses on how to compose objects into tree structures which represent part-whole hierarchies with arbitrary depth and width. The key idea in the Composite Pattern is to treat individual objects and compositions of these objects uniformly, so that the client can simply use the same code regardless of whether it is a leaf node or not in the tree structure. For example, a software system consists of subsystems which are further composed of either other subsystems or classes. Here a class is a leaf node and a subsystem

is a composite node. Another example is the folder in the file system. A folder can contain folders or files. A file is a leaf node and a folder is a composite node in the tree structure.

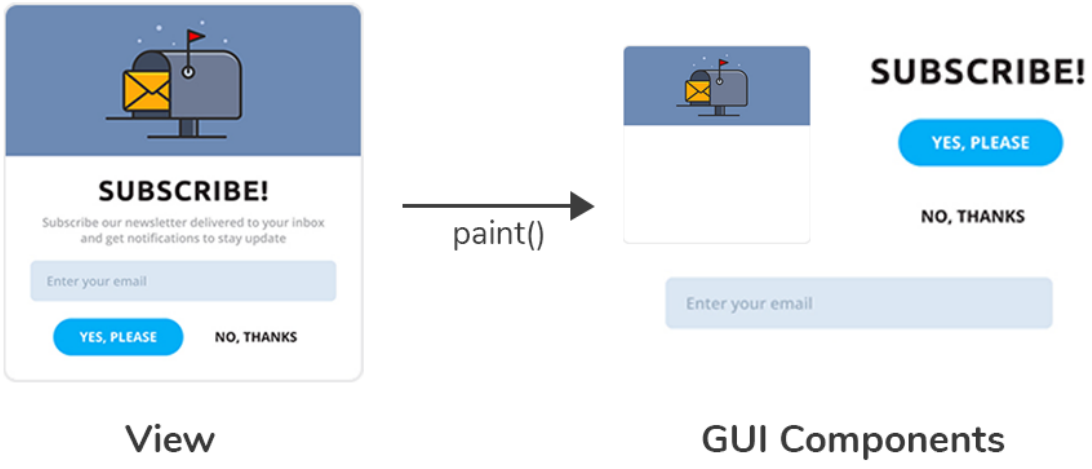
The following class diagram shows the composite pattern. To treat the leaf and composite uniformly, a Component interface or abstract class is used as the base class for both leaf and composite class. The Composite class consists of components which can be either Leaves or Composites.



Component Pattern Class Diagram (Gamma, E et al., 1994)

Let us use the filesystem example again. A folder can contain files or other folders. To list or find the files in a folder, we need to list or search all subfolders recursively. The diagram below shows how to use the composite pattern to solve this problem.

COMPOSITE

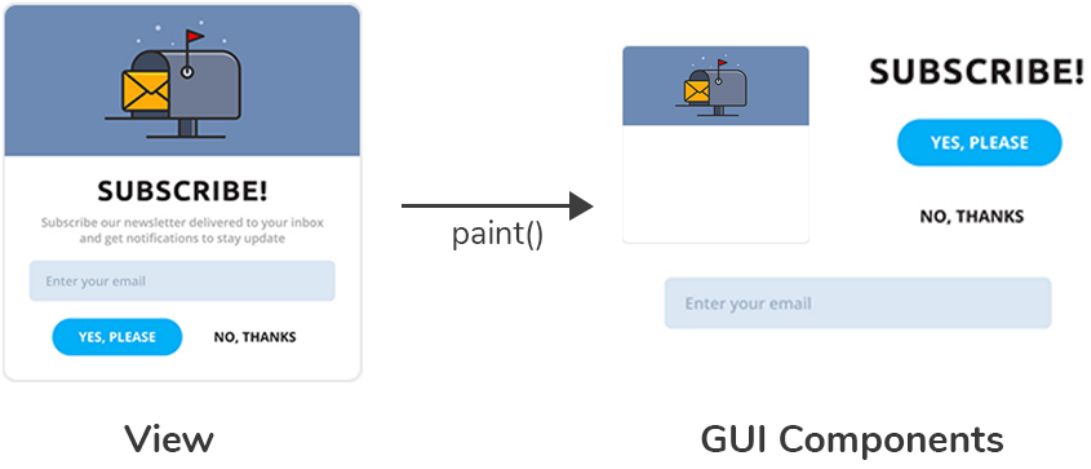


The view is a composite of GUI components (labels, buttons, text entry, etc.). The top-level component contains other components, which contain other components and so on until you get to the leaf nodes.

File System Class Diagram using Composite Pattern

The composite pattern is often used in the views in MVC. A screen is usually a window that may contain subwindows or basic view components such as buttons, text fields, and labels. For example, Java Swing Library has a JComponent which is an abstract base class for both leaf components such as JLabel and the composite class such as Jpanel. In Android, the View class is the abstract base class for both leaf components such as TextView and the container component such as ViewGroup. To draw a container GUI component, it will draw all its children components recursively.

COMPOSITE



The view is a composite of GUI components (labels, buttons, text entry, etc.). The top-level component contains other components, which contain other components and so on until you get to the leaf nodes.

(Based on: Freeman, E et al., 2004)

While design patterns provide shared vocabulary and solutions for some common problems to improve flexibility and extensibility. , one should be aware that improper use of design patterns can counteract its benefit. Sometimes the cost of additional complexity as well as processing and memory overhead may outweigh the benefits. It requires a realistic understanding of how the software is likely to change.

Test Yourself

What does DRY stand for?

Don't Repeat Yourself: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. A modification of any single element of a system does not require a change in other logically unrelated elements

Test Yourself

What does SOLID stand for?

- **S**ingle responsibility principle
- **O**pen–closed principle

- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Test Yourself

Give an example of using specification inheritance, implementation inheritance and implementation reuse through composition respectively.

STOP AND CONSIDER

Choose the right pattern for the following cases:

1. Define a family of algorithms, and encapsulate each algorithm to make entire algorithms interchangeable at the run time.

○ Observer ○ Strategy ○ Composite

2. Defines a one-to-many relationship between an object and other objects that are notified about changes in its state.

○ Observer ○ Strategy ○ Composite

3. Treat entire whole-part hierarchies uniformly.

○ Observer ○ Strategy ○ Composite

Check Answers | Download Correct Answers

Conclusion

In this module, we discussed various issues and activities in software design. We emphasized the importance of setting design goals and following design principles. We also introduced several architecture styles and design patterns.

Bibliography

- Ingeno, J. (2018) Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts.
- Fielding, R. T. (2000) . Architectural styles and the design of network-based software architectures. Dissertation. University of California, Irvine.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J.. (1994) Design patterns, software engineering, object-oriented programming. Addison-Wesley.
- Freeman, E., Bates, B., Sierra, K., Robson, E.. (2004). Head First Design Patterns: A Brain-Friendly Guide. O'Reilly Media.