Design Patterns

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Uses of Design Patterns

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

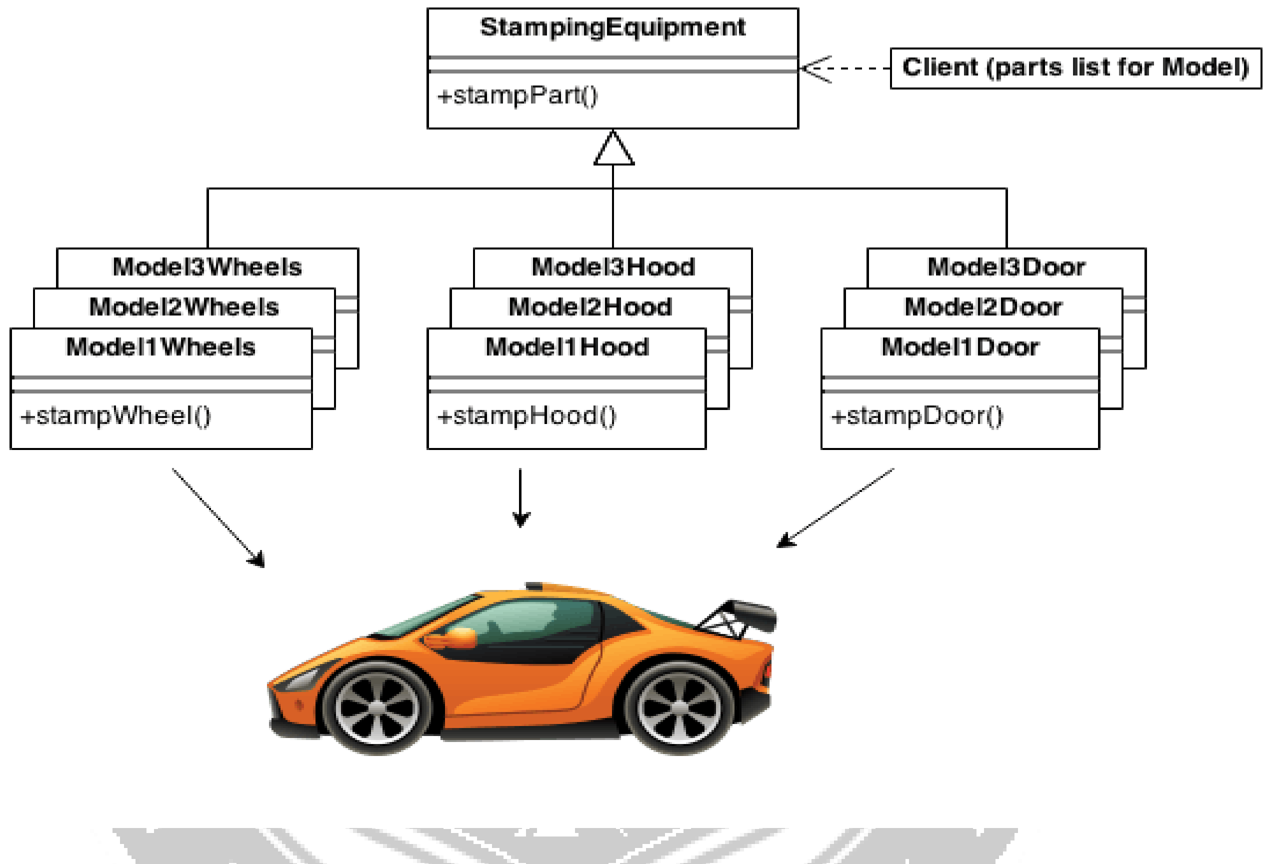
Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

Creational design patterns

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While

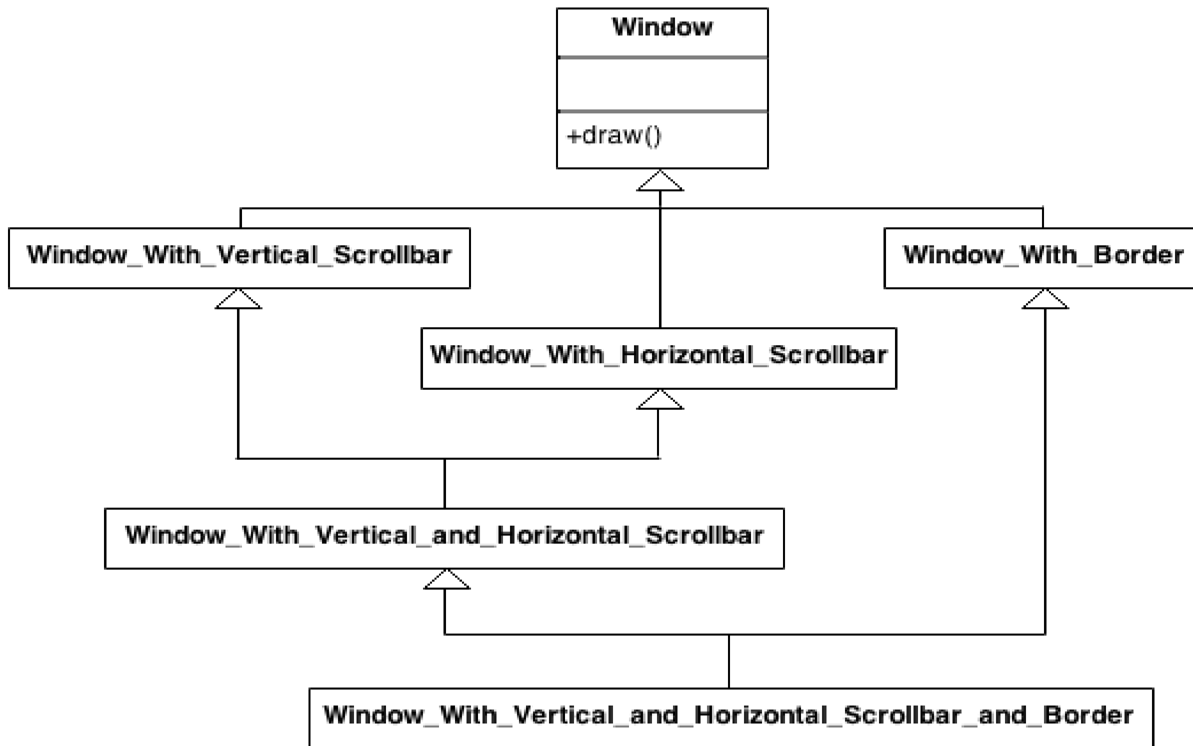
class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.



- **AbstractFactory**
  Creates an instance of several families of classes
- **Builder**
  Separates object construction from its representation
- **FactoryMethod**
  Creates an instance of several derived classes
- **ObjectPool**
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- **Prototype**
  A fully initialized instance to be copied or cloned
- **Singleton**
  A class of which only a single instance can exist

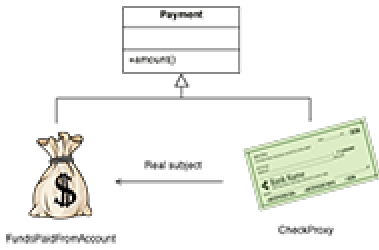
Structural design patterns

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality



ty.

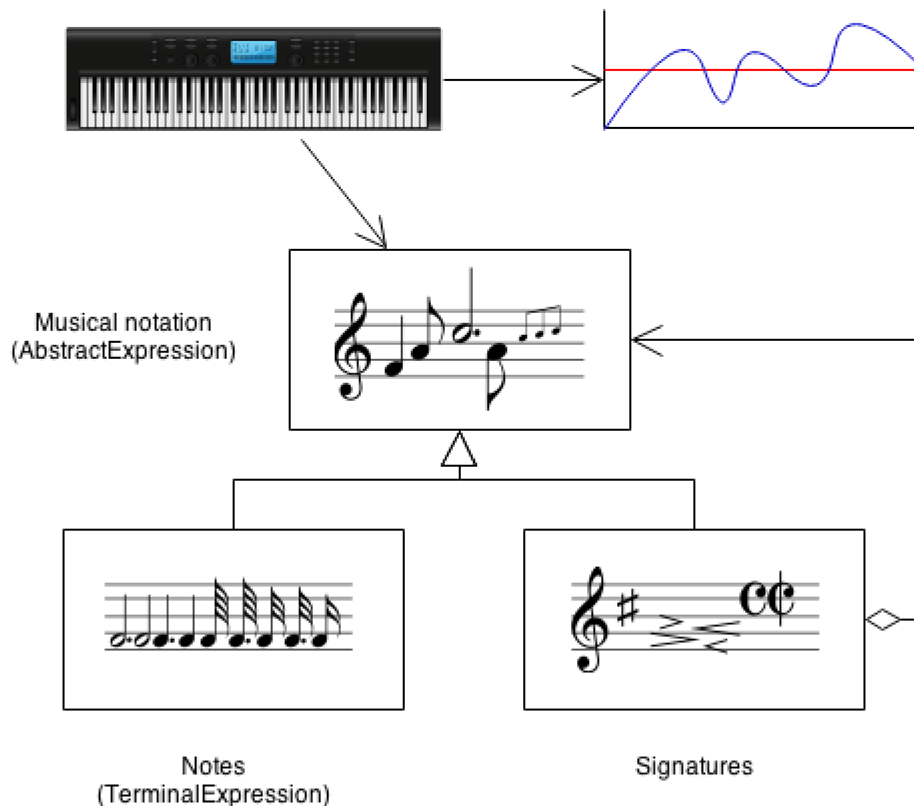
- **Adapter**
  Match interfaces of different classes
- **Bridge**
  Separates an object's interface from its implementation
- **Composite**
  A tree structure of simple and composite objects
- **Decorator**
  Add responsibilities to objects dynamically
- **Facade**
  A single class that represents an entire subsystem

- **Flyweight**
  A fine-grained instance used for efficient sharing
  

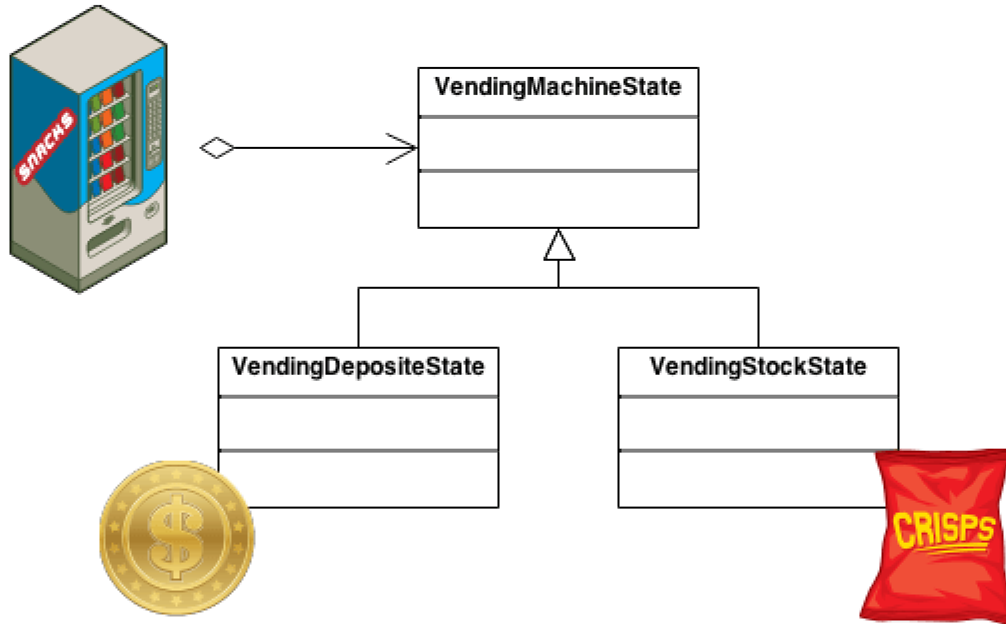
- **PrivateClassData**
  Restricts accessor/mutator access
- **Proxy**
  An object representing another object

Behavioral design patterns

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.



- **Chainofresponsibility**
  A way of passing a request between a chain of objects
- **Command**
  Encapsulate a command request as an object
- **Interpreter**
  A way to include language elements in a program
- **Iterator**
  Sequentially access the elements of a collection
- **Mediator**
  Defines simplified communication between classes
- **Memento**
  Capture and restore an object's internal state
- **NullObject**
  Designed to act as a default value of an object
- **Observer**
  A way of notifying change to a number of classes



- **State**
  Alter an object's behavior when its state changes
- **Strategy**
  Encapsulates an algorithm inside a class
- **Templatemethod**
  Defer the exact steps of an algorithm to a subclass
- **Visitor**
  Defines a new operation to a class without change

Types of Design Patterns

There are three types of Design Patterns,

- Creational Design Pattern
- Structural Design Pattern
- Behavioral Design Pattern

Creational Design Pattern

Creational Design Pattern abstract the instantiation process. They help in making a system independent of how its objects are created, composed and represented.

Importance of Creational Design Patterns:

- A class creational Pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.
- Creational patterns become important as systems evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hardcoding a

fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones.
- Creating objects with particular behaviors requires more than simply instantiating a class.

When to ue Creational Design Patterns

- **Complex Object Creation:** Use creational patterns when the process of creating an object is complex, involving multiple steps, or requires the configuration of various parameters.
- **Promoting Reusability:** Creational patterns promote object creation in a way that can be reused across different parts of the code or even in different projects, enhancing modularity and maintainability.
- **Reducing Coupling:** Creational patterns can help reduce the coupling between client code and the classes being instantiated, making the system more flexible and adaptable to changes.
- **Singleton Requirements:** Use the Singleton pattern when exactly one instance of a class is needed, providing a global point of access to that instance.
- **Step-by-Step Construction:** Builder pattern of creational design patterns is suitable when you need to construct a complex object step by step, allowing for the creation of different representations of the same object.

Advantages of Creational Design Patterns

- **Flexibility and Adaptability:** Creational patterns make it easier to introduce new types of objects or change the way objects are created without modifying existing client code. This enhances the system’s flexibility and adaptability to change.
- **Reusability:** By providing a standardized way to create objects, creational patterns promote code reuse across different parts of the application or even in different projects. This leads to more maintainable and scalable software.
- **Centralized Control:** Creational patterns, such as Singleton and Factory patterns, allow for centralized control over the instantiation process. This can be advantageous in managing resources, enforcing constraints, or ensuring a single point of access.
- **Scalability:** With creational patterns, it’s easier to scale and extend a system by adding new types of objects or introducing variations without causing major disruptions to the existing codebase.
- **Promotion of Good Design Practices:** Creational patterns often encourage adherence to good design principles such as abstraction, encapsulation, and the separation of concerns. This leads to cleaner, more maintainable code.

Disadvantages of Creational Design Patterns

- **Increased Complexity:** Introducing creational patterns can sometimes lead to increased complexity in the codebase, especially when dealing with a large number of classes, interfaces, and relationships.
- **Overhead:** Using certain creational patterns, such as the Abstract Factory or Prototype pattern, may introduce overhead due to the creation of a large number of classes and interfaces.
- **Dependency on Patterns:** Over-reliance on creational patterns can make the codebase dependent on a specific pattern, making it challenging to adapt to changes or switch to alternative solutions.

- **Readability and Understanding:** The use of certain creational patterns might make the code less readable and harder to understand, especially for developers who are not familiar with the specific pattern being employed.

Structural Design Patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations.

Importance of Structural Design Patterns

- This pattern is particularly useful for making independently developed class libraries work together.
- Structural object patterns describe ways to compose objects to realize new functionality.
- It added flexibility of object composition comesfrom the ability to change the composition at run-time, which is impossible with static class composition.

When to ue Structural Design Patterns

- **Adapting to Interfaces:** Use structural patterns like the Adapter pattern when you need to make existing classes work with others without modifying their source code. This is particularly useful when integrating with third-party libraries or legacy code.
- **Organizing Object Relationships:** Structural patterns such as the Decorator pattern are useful when you need to add new functionalities to objects by composing them in a flexible and reusable way, avoiding the need for subclassing.
- **Simplifying Complex Systems:** When dealing with complex systems, structural patterns like the Facade pattern can be used to provide a simplified and unified interface to a set of interfaces in a subsystem.
- **Managing Object Lifecycle:** The Proxy pattern is helpful when you need to control access to an object, either for security purposes, to delay object creation, or to manage the object's lifecycle.
- **Hierarchical Class Structures:** The Composite pattern is suitable when dealing with hierarchical class structures where clients need to treat individual objects and compositions of objects uniformly.

Advantages of Structural Design Patterns

- **Flexibility and Adaptability:** Structural patterns enhance flexibility by allowing objects to be composed in various ways. This makes it easier to adapt to changing requirements without modifying existing code.
- **Code Reusability:** These patterns promote code reuse by providing a standardized way to compose objects. Components can be reused in different contexts, reducing redundancy and improving maintainability.
- **Improved Scalability:** As systems grow in complexity, structural patterns provide a scalable way to organize and manage the relationships between classes and objects. This supports the growth of the system without causing a significant increase in complexity.
- **Simplified Integration:** Structural patterns, such as the Adapter pattern, facilitate the integration of existing components or third-party libraries by providing a standardized interface. This makes it easier to incorporate new functionalities into an existing system.

- **Easier Maintenance:** By promoting modularity and encapsulation, structural patterns contribute to easier maintenance. Changes to one part of the system are less likely to affect other parts, reducing the risk of unintended consequences.
- **Solves Recurring Design Problems:** These patterns encapsulate solutions to recurring design problems. By applying proven solutions, developers can focus on higher-level design challenges unique to their specific applications.

Disadvantages of Structural Design Patterns

- **Complexity:** Introducing structural patterns can sometimes lead to increased complexity in the codebase, especially when multiple patterns are used or when dealing with a large number of classes and interfaces.
- **Overhead:** Some structural patterns, such as the Composite pattern, may introduce overhead due to the additional layers of abstraction and complexity introduced to manage hierarchies of objects.
- **Maintenance Challenges:** Changes to the structure of classes or relationships between objects may become more challenging when structural patterns are heavily relied upon. Modifying the structure may require updates to multiple components.
- **Limited Applicability:** Not all structural patterns are universally applicable. The suitability of a pattern depends on the specific requirements of the system, and using a pattern in the wrong context may lead to unnecessary complexity.

Behavioral Design Pattern

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.

Importance of Behavioral Design Pattern

- These patterns characterize complex control flow that's difficult to follow at run-time.
- They shift focus away from flow of control to let you concentratejust on the way objects are interconnected.
- Behavioral class patterns use inheritance to distribute behavior between classes.

When to ue Behavioral Design Patterns

- **Communication Between Objects:** Use behavioral patterns when you want to define how objects communicate, collaborate, and interact with each other in a flexible and reusable way.
- **Encapsulation of Behavior:** Apply behavioral patterns to encapsulate algorithms, strategies, or behaviors, allowing them to vary independently from the objects that use them. This promotes code reusability and maintainability.
- **Dynamic Behavior Changes:** Use behavioral patterns when you need to allow for dynamic changes in an object's behavior at runtime without altering its code. This is particularly relevant for systems that require flexibility in behavior.
- **State-Dependent Behavior:** State pattern is suitable when an object's behavior depends on its internal state, and the object needs to change its behavior dynamically as its state changes.
- **Interactions Between Objects:** Behavioral patterns are valuable when you want to model and manage interactions between objects in a way that is clear, modular, and easy to understand.

Advantages of Behavioral Design Patterns

Flexibility and Adaptability:

- Behavioral patterns enhance flexibility by allowing objects to interact in a more dynamic and adaptable way. This makes it easier to modify or extend the behavior of a system without altering existing code.
- **Code Reusability:**
- Behavioral patterns promote code reusability by encapsulating algorithms, strategies, or behaviors in separate objects. This allows the same behavior to be reused across different parts of the system.
- **Separation of Concerns:**
- These patterns contribute to the separation of concerns by dividing the responsibilities of different classes, making the codebase more modular and easier to understand.
- **Encapsulation of Algorithms:**
- Behavioral patterns encapsulate algorithms, strategies, or behaviors in standalone objects, making it possible to modify or extend the behavior without affecting the client code.
- **Ease of Maintenance:**
- With well-defined roles and responsibilities for objects, behavioral patterns contribute to easier maintenance. Changes to the behavior can be localized, reducing the impact on the rest of the code.

Disadvantages of Behavioral Design Patterns

- **Increased Complexity:** Introducing behavioral patterns can sometimes lead to increased complexity in the codebase, especially when multiple patterns are used or when there is an excessive use of design patterns in general.
- **Over-Engineering:** There is a risk of over-engineering when applying behavioral patterns where simpler solutions would suffice. Overuse of patterns may result in code that is more complex than necessary.
- **Limited Applicability:** Not all behavioral patterns are universally applicable. The suitability of a pattern depends on the specific requirements of the system, and using a pattern in the wrong context may lead to unnecessary complexity.
- **Code Readability:** In certain cases, applying behavioral patterns may make the code less readable and harder to understand, especially for developers who are not familiar with the specific pattern being employed.
- **Scalability Concerns:** As the complexity of a system increases, the scalability of certain behavioral patterns may become a concern. For example, the Observer pattern may become less efficient with a large number of observers.

Model-view-controller (MVC)

Definition

The MVC pattern in Software Engineering Architecture is defined as an application being separated into three logical components: Model, View and Controller.

Model

This component in the architecture will represent all data-related logic. This includes defining how the data is formed. In other words, this holds the definition for many of the types that we use in the application. In many cases, the model here refers to the type of data that we are dealing with in the application. This component also notifies its dependents about data changes.

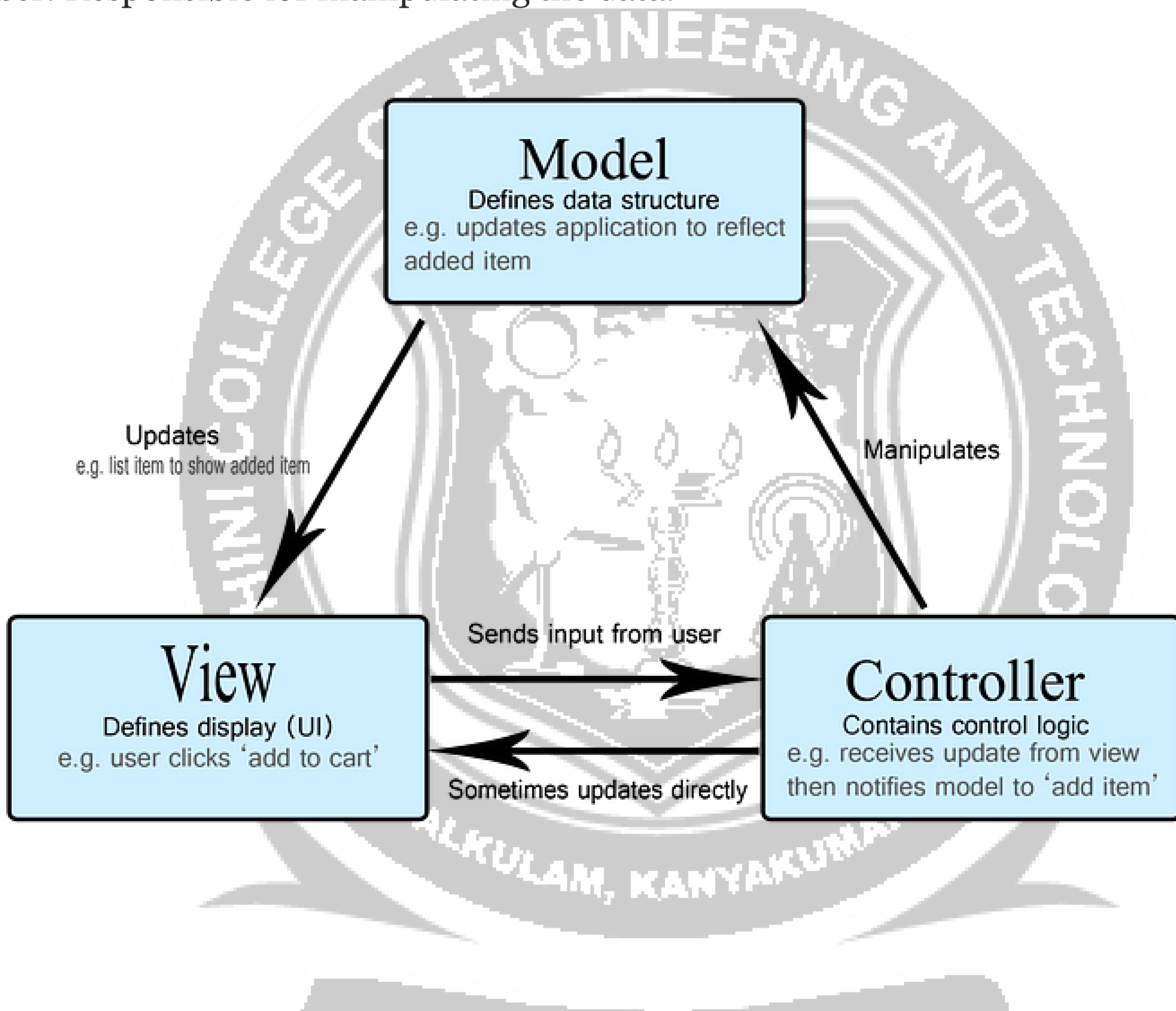
View

Contains all User interface (UI) logic in the application. This component of the application encapsulates mainly the UI related logic which includes things that the end user will manipulate like dropdown buttons and web pages etc.

Controller

Controllers exist as a layer between Model and View components to process all the business logic arising from user input. It is responsible to handle inputs from the View

components, manipulate data using the models from the Model component and then finally interact with the view components again to render the final output to the end user. Responsible for manipulating the data.



Why MVC

The MVC pattern today is widely used for many applications and remain a popular choice. This is due to a few key reasons

Faster Development Time

Given the separation of the applications into the three distinct areas, this means that more developers are able to work on each part separately. e.g if a developer works on the model, he is not directly blocking another developer from building up the view component of the application and thus allows teams to speed up development purely due to the nature of the architecture itself.

Greater Testability

Each component being separated from each other means that developers are able to test each one separately and in isolation. This is made easier due to the clear separation of concerns that is applied in this architecture pattern. e.g a Model can be tested easily without the view component.

Easy extension of views/modification

Any changes in view component will usually not affect the model component, hence developers using this pattern can easily extend and add new views to the application to display the data from model in different ways. Thus, modifications are easier to be isolated to a single component instead of affecting the entire application

MVC in Real World application

Web applications

Many web applications today run primarily on MVC architecture. In particular, ASP.NET MVC framework offers and MVC pattern as one of the development model.

This framework provides developers with a MVC abstraction built on top of ASP.NEXT and thus provide a large set of added functionality.
Sample code from ASP.NET documentation

An example of a controller action in ASP.NET MVC framework.

Another example framework for web that uses MVC is Sails. Sails is a nodeJs framework that provides added functionality. A sails app comes preconfigured with the MVC structure predefined and developers can just use it right out of the box.

PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD