Design Patterns

Prof. Michele Loreti

Programmazione Avanzata
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Big questions

What is a design pattern?

Big questions

What is a design pattern?

What is the advantage of knowing/using design patterns?

Big questions

What is a design pattern?

What is the advantage of knowing/using design patterns?

Which patterns are named in the reading?

Big questions

What is a design pattern?

What is the advantage of knowing/using design patterns?

Which patterns are named in the reading?

What are the key ideas of those patterns?

Design challenges

Designing software for reuse is hard.

Design challenges

Designing software for reuse is hard. One must find:

- a good problem decomposition, and the right software;

Design challenges

Designing software for reuse is hard. One must find:

- a good problem decomposition, and the right software;
- a design with flexibility, modularity and elegance.

Design challenges

Designing software for reuse is hard. One must find:

- a good problem decomposition, and the right software;
- a design with flexibility, modularity and elegance.

Design challenges

Designing software for reuse is hard. One must find:

- a good problem decomposition, and the right software;
- a design with flexibility, modularity and elegance.

Designs often emerge from trial and error.

Design challenges

Designing software for reuse is hard. One must find:

- a good problem decomposition, and the right software;
- a design with flexibility, modularity and elegance.

Designs often emerge from trial and error.

Successful designs do exist

- two designs they are almost never identical;
- they exhibit some recurring characteristics.

Design challenges

Designing software for reuse is hard. One must find:

- a good problem decomposition, and the right software;
- a design with flexibility, modularity and elegance.

Designs often emerge from trial and error.

Successful designs do exist

- two designs they are almost never identical;
- they exhibit some recurring characteristics.

Can designs be described, codified or standardised?

- this would short circuit the trial and error phase;
- produce "better" software faster.

Design Patterns

Design Pattern: a solution to a common software problem in a context

Design Patterns

Design Pattern: a solution to a common software problem in a context

- describes a recurring software structure;

Design Patterns

Design Pattern: a solution to a common software problem in a context

- describes a recurring software structure;
- is abstract from programming language;

Design Patterns

Design Pattern: a solution to a common software problem in a context

- describes a recurring software structure;
- is abstract from programming language;
- identifies classes and their roles in the solution to a problem;

Design Patterns

Design Pattern: a solution to a common software problem in a context

- describes a recurring software structure;
- is abstract from programming language;
- identifies classes and their roles in the solution to a problem;
- patterns are not code or designs; must be instantiated/applied.

Design Patterns

Design Pattern: a solution to a common software problem in a context

- describes a recurring software structure;
- is abstract from programming language;
- identifies classes and their roles in the solution to a problem;
- patterns are not code or designs; must be instantiated/applied.

Design Patterns

Design Pattern: a solution to a common software problem in a context

- describes a recurring software structure;
- is abstract from programming language;
- identifies classes and their roles in the solution to a problem;
- patterns are not code or designs; must be instantiated/applied.

Example: Iterator pattern!

Design Patterns

Design Pattern: a solution to a common software problem in a context

- describes a recurring software structure;
- is abstract from programming language;
- identifies classes and their roles in the solution to a problem;
- patterns are not code or designs; must be instantiated/applied.

Example: Iterator pattern! The Iterator pattern defines an interface that declares methods for sequentially accessing the objects in a collection.

History of patterns

The concept of a pattern was first expressed in Christopher Alexander's work *A Pattern Language* in 1977 (2543 patterns).

History of patterns

The concept of a pattern was first expressed in Christopher Alexander's work *A Pattern Language* in 1977 (2543 patterns).

In 1990 a group called the Gang of Four or GoF (Gamma, Helm, Johnson, Vlissides) compile a catalog of design patterns.

History of patterns

The concept of a pattern was first expressed in Christopher Alexander's work *A Pattern Language* in 1977 (2543 patterns).

In 1990 a group called the Gang of Four or GoF (Gamma, Helm, Johnson, Vlissides) compile a catalog of design patterns.

In 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software*, which is a classic of the field, is published.

Benefits of using patterns

Patterns are a common design vocabulary.

- allows to abstract a problem and talk about that abstraction in isolation from its implementation;
- embodies a culture; domain-specific patterns increase design speed.

Benefits of using patterns

Patterns are a common design vocabulary.

- allows to abstract a problem and talk about that abstraction in isolation from its implementation;
- embodies a culture; domain-specific patterns increase design speed.

Patterns capture design expertise and allow that expertise to be communicated. **Promotes design reuse and avoid mistakes**.

Benefits of using patterns

Patterns are a common design vocabulary.

- allows to abstract a problem and talk about that abstraction in isolation from its implementation;
- embodies a culture; domain-specific patterns increase design speed.

Patterns capture design expertise and allow that expertise to be communicated. **Promotes design reuse and avoid mistakes**.

Improve documentation (less is needed) and **understandability** (patterns are described well once).

Gang of Four (GoF) patterns

Creational Patterns: abstracting the object-instantiation process.

- Factory Method, Abstract Factory, Singleton, Builder, Prototype.

Gang of Four (GoF) patterns

Creational Patterns: abstracting the object-instantiation process.

- Factory Method, Abstract Factory, Singleton, Builder, Prototype.

Structural Patterns: how objects/classes can be combined to form larger structures.

- Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.

Gang of Four (GoF) patterns

Creational Patterns: abstracting the object-instantiation process.

- Factory Method, Abstract Factory, Singleton, Builder, Prototype.

Structural Patterns: how objects/classes can be combined to form larger structures.

- Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.

Behavioral Patterns: communication between objects.

- Command, Interpreter, Iterator, Mediator, Observer, State, Strategy, Chain of Responsibility, Visitor, Template Method.

Factory Method

The Factory Method design pattern solves problems like:

- How can an object be created so that subclasses can redefine which class to instantiate?

Factory Method

The Factory Method design pattern solves problems like:

- How can an object be created so that subclasses can redefine which class to instantiate?
- How can a class defer instantiation to subclasses?

Factory Method

The Factory Method design pattern solves problems like:

- How can an object be created so that subclasses can redefine which class to instantiate?
- How can a class defer instantiation to subclasses?

Factory Method

The Factory Method design pattern solves problems like:

- How can an object be created so that subclasses can redefine which class to instantiate?
- How can a class defer instantiation to subclasses?

Creating an object directly within the class that requires (uses) the object is inflexible:

- it commits the class to a particular object;
- it impossible to change the instantiation independently from (without having to change) the class.

Factory Method

The Factory Method design pattern solves problems like:

- How can an object be created so that subclasses can redefine which class to instantiate?
- How can a class defer instantiation to subclasses?

Creating an object directly within the class that requires (uses) the object is inflexible:

- it commits the class to a particular object;
- it impossible to change the instantiation independently from (without having to change) the class.

The Factory Method design pattern describes how to solve such problems:

- Define a separate operation (factory method) for creating an object.
- Create an object by calling a factory method.

Factory Method

Definition: Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses.

Factory Method

Definition: Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses.

Problem to solve: handle the process creation process

- reduce code duplication;
- provide information not accessible to the composing object;
- abstract the creation process.

Factory Method

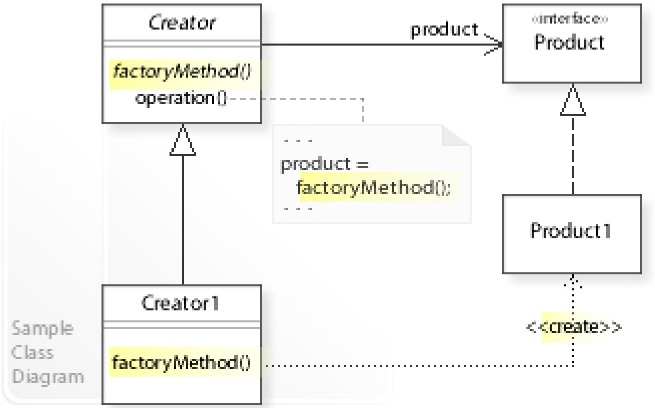
Definition: Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses.

Problem to solve: handle the process creation process

- reduce code duplication;
- provide information not accessible to the composing object;
- abstract the creation process.

The factory method pattern relies on inheritance, as object creation is delegated to subclasses that implement the factory method to create objects.

Factory Method



Abstract Factory

Abstract Factory

The Abstract Factory design pattern solves problems like:

- How can an application be independent of how its objects are created?
- How can a class be independent of how the objects it requires are created?
- How can families of related or dependent objects be created?

Abstract Factory

The Abstract Factory design pattern solves problems like:

- How can an application be independent of how its objects are created?
- How can a class be independent of how the objects it requires are created?
- How can families of related or dependent objects be created?

This pattern:

- Encapsulate object creation in a separate (factory) object defined via an interface (AbstractFactory);
- A class delegates object creation to a factory object instead of creating objects directly.

Abstract Factory

Definition: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Abstract Factory

Definition: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Usage: The factory determines the actual concrete type of object to be created, and it is here that the object is actually created.

Abstract Factory

Definition: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Usage: The factory determines the actual concrete type of object to be created, and it is here that the object is actually created.

However, the factory only returns an abstract pointer to the created concrete object.

Abstract Factory

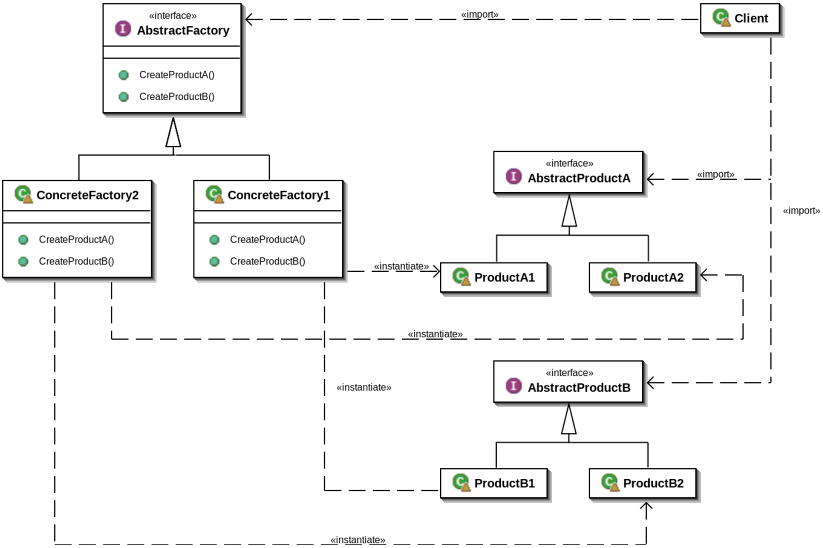
Definition: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Usage: The factory determines the actual concrete type of object to be created, and it is here that the object is actually created.

However, the factory only returns an abstract pointer to the created concrete object.

This insulates client code from object creation by having clients ask a factory object to create an object of the desired abstract type and to return an abstract pointer to the object.

Abstract Factory



Singleton Pattern

The singleton design pattern solves problems like:

- How can it be ensured that a class has only one instance?
- How can the sole instance of a class be accessed easily?
- How can a class control its instantiation?
- How can the number of instances of a class be restricted?

Singleton Pattern

The singleton design pattern solves problems like:

- How can it be ensured that a class has only one instance?
- How can the sole instance of a class be accessed easily?
- How can a class control its instantiation?
- How can the number of instances of a class be restricted?

The singleton design pattern describes how to solve such problems:

- Hide the constructor of the class.
- Define a public static operation (getInstance()) that returns the sole instance of the class.

Singleton Pattern

The key idea in this pattern is to make the class itself responsible for controlling its instantiation (that it is instantiated only once).

Singleton Pattern

The key idea in this pattern is to make the class itself responsible for controlling its instantiation (that it is instantiated only once).

The hidden constructor (declared private) ensures that the class can never be instantiated from outside the class.

Singleton Pattern

The key idea in this pattern is to make the class itself responsible for controlling its instantiation (that it is instantiated only once).

The hidden constructor (declared private) ensures that the class can never be instantiated from outside the class.

The public static operation can be accessed easily by using the class name and operation name (`Singleton.getInstance()`).

Singleton Pattern

```
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton()
    ;

    private Singleton() {
      //If needed, parameters can be read from local context!
    }

    public static Singleton getInstance() {
        return INSTANCE;
    }

    ....
}
```

Composite pattern

What problems can the Composite design pattern solve?

- A part-whole hierarchy should be represented so that clients can treat part and whole objects uniformly.
- A part-whole hierarchy should be represented as tree structure.

Composite pattern

What problems can the Composite design pattern solve?

- A part-whole hierarchy should be represented so that clients can treat part and whole objects uniformly.
- A part-whole hierarchy should be represented as tree structure.

What solution does the Composite design pattern describe?

- Define a unified Component interface for both part (Leaf) objects and whole (Composite) objects.
- Individual Leaf objects implement the Component interface directly
- Composite objects forward requests to their child components.

Composite Pattern

Composite enables clients to work through the Component interface to treat Leaf and Composite objects uniformly:

- Leaf objects perform a request directly;
- Composite objects forward the request to their child components recursively downwards the tree structure.

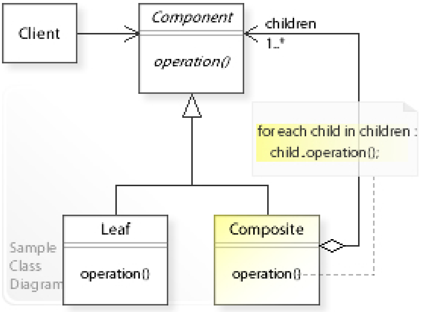
Composite Pattern

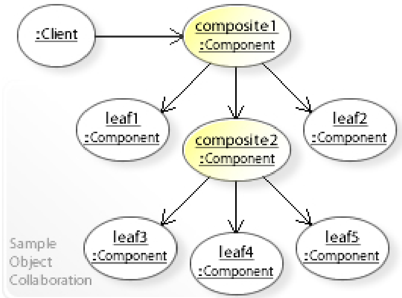
Composite enables clients to work through the Component interface to treat Leaf and Composite objects uniformly:

- Leaf objects perform a request directly;
- Composite objects forward the request to their child components recursively downwards the tree structure.

This makes client classes easier to implement, change, test, and reuse.

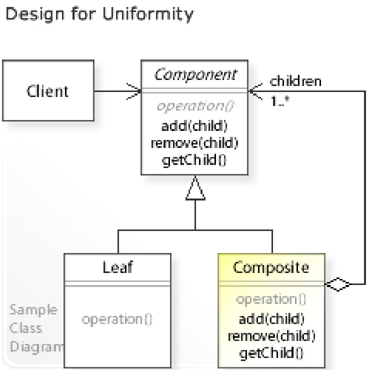
Composite Pattern

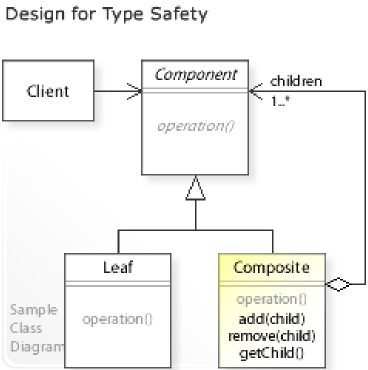




Composite Pattern

Two variants. . .





Composite Pattern
Example...

We can consider a set of classes modelling a figure.

Composite Pattern

Example...

We can consider a set of classes modelling a figure.

Each figure can be:

- a basic figure: Rectangle, Ellipse , Triangle .
- a group of figures: Group.

Composite Pattern

Example. . .

We can consider a set of classes modelling a figure.

Each figure can be:

- a basic figure: Rectangle, Ellipse, Triangle.
- a group of figures: Group.

Operation is:

- `draw( Graphics g , int x , int y )`

Composite Pattern

Java code. . .

Decorator Pattern

What problems can the Decorator design pattern solve?

Decorator Pattern

What problems can the Decorator design pattern solve?

- Responsibilities should be added to (and removed from) an object dynamically at run-time.

Decorator Pattern

What problems can the Decorator design pattern solve?

- Responsibilities should be added to (and removed from) an object dynamically at run-time.
- A flexible alternative to subclassing for extending functionality should be provided.

Decorator Pattern

What problems can the Decorator design pattern solve?

- Responsibilities should be added to (and removed from) an object dynamically at run-time.
- A flexible alternative to subclassing for extending functionality should be provided.
- When using subclassing, different subclasses extend a class in different ways. But an extension is bound to the class at compile-time and can't be changed at run-time.

Decorator Pattern

What solution does the Decorator design pattern describe?

Decorator Pattern

What solution does the Decorator design pattern describe? Define Decorator objects that:

- implement the interface of the extended (decorated) object (Component) transparently by forwarding all requests to it and perform additional functionality before/after forwarding a request.
- This enables to work through different Decorator objects to extend the functionality of an object dynamically at run-time.

Decorator Pattern

Decorator Pattern is based on the following sequence of steps:

Decorator Pattern

Decorator Pattern is based on the following sequence of steps:

- Subclass the original Component class into a Decorator class;

Decorator Pattern

Decorator Pattern is based on the following sequence of steps:

- Subclass the original Component class into a Decorator class;
- In the Decorator class, add a Component pointer as a field;

Decorator Pattern

Decorator Pattern is based on the following sequence of steps:

- Subclass the original Component class into a Decorator class;
- In the Decorator class, add a Component pointer as a field;
- In the Decorator class, pass a Component to the Decorator constructor to initialise the Component pointer;

Decorator Pattern

Decorator Pattern is based on the following sequence of steps:

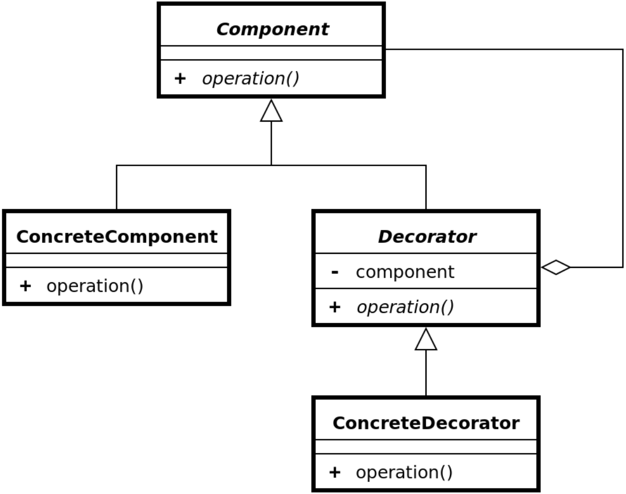
- Subclass the original Component class into a Decorator class;
- In the Decorator class, add a Component pointer as a field;
- In the Decorator class, pass a Component to the Decorator constructor to initialise the Component pointer;
- In the Decorator class, forward all Component methods to the Component pointer;

Decorator Pattern

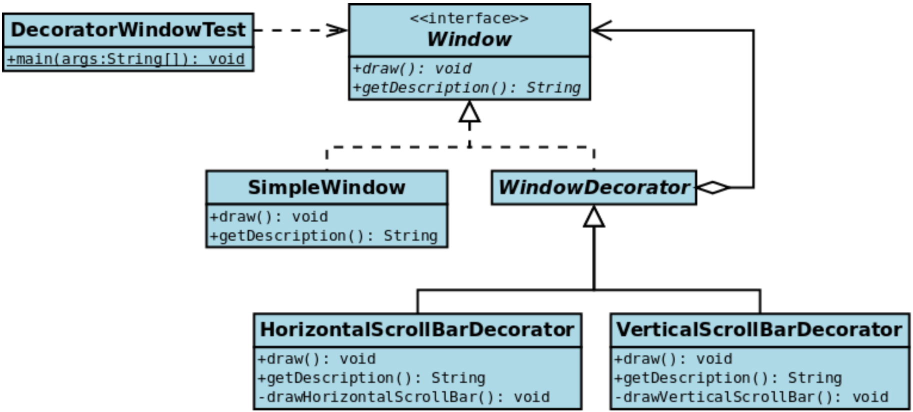
Decorator Pattern is based on the following sequence of steps:

- Subclass the original Component class into a Decorator class;
- In the Decorator class, add a Component pointer as a field;
- In the Decorator class, pass a Component to the Decorator constructor to initialise the Component pointer;
- In the Decorator class, forward all Component methods to the Component pointer;
- and In the Decorator class, override any Component method(s) whose behaviour needs to be modified.

Decorator Pattern



Decorator Pattern



Command

What solution does the Command design pattern describe?

- Define separate (command) objects that encapsulate a request.

This enables one to configure a class with a command object that is used to perform a request. The class is no longer coupled to a particular request and has no knowledge (is independent) of how the request is carried out.

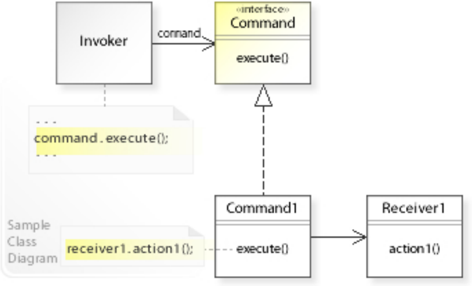
Command

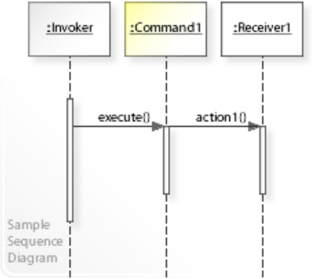
What solution does the Command design pattern describe?

- Define separate (command) objects that encapsulate a request.
- A class delegates a request to a command object instead of implementing a particular request directly.

This enables one to configure a class with a command object that is used to perform a request. The class is no longer coupled to a particular request and has no knowledge (is independent) of how the request is carried out.

Command


Invoker
comnand
«interface»
Command
execute()
...
command.execute();
...
Command1
execute()
Receiver1
action1()
Sample
Class
Diagram
receiver1.action1();


:Invoker
:Command1
:Receiver1
execute()
action1()
Sample
Sequence
Diagram

Observer

What problems can the Observer design pattern solve?

What problems can the Observer design pattern solve?

- A one-to-many dependency between objects should be defined without making the objects tightly coupled.
- It should be ensured that when one object changes state an open-ended number of dependent objects are updated automatically.
- It should be possible that one object can notify an open-ended number of other objects.

Observer

What solution does the Observer design pattern describe?

Observer

What solution does the Observer design pattern describe?

- Define `Observable` and Observer objects.
- When a subject changes state, all registered observers are notified and updated automatically.

Observable

Observable

This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.

Observable

This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.

An observable object can have one or more observers. An observer may be any object that implements interface Observer.

Observable

This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.

An observable object can have one or more observers. An observer may be any object that implements interface Observer.

After an observable instance changes, an application calling the Observable's `notifyObservers` method causes all of its observers to be notified of the change by a call to their update method.

Observable

Methods. . .

```
void addObserver(Observer o)

protected void clearChanged()

int countObservers()

void deleteObserver(Observer o)

void deleteObservers()

boolean hasChanged()

void notifyObservers()

void notifyObservers(Object arg)

protected void setChanged()
```

Observer

A class can implement the Observer interface when it wants to be informed of changes in observable objects.

Observer

A class can implement the Observer interface when it wants to be informed of changes in observable objects.

This interface provides a single method:

```
void update(Observable o, Object arg)
```

Observer

A class can implement the Observer interface when it wants to be informed of changes in observable objects.

This interface provides a single method:

```
void update(Observable o, Object arg)
```

This method is called whenever the observed object is changed. An application calls an Observable object's notifyObservers method to have all the object's observers notified of the change.

MVC: Model View Controller

Model-view-controller is commonly used for developing software that divides an application into three interconnected parts:

- Model;
- View;
- Controller.

MVC: Model View Controller

Model-view-controller is commonly used for developing software that divides an application into three interconnected parts:

- Model;
- View;
- Controller.

This is done to separate internal representations of information from the ways information is presented to and accepted from the user.

MVC: Model View Controller

Model-view-controller is commonly used for developing software that divides an application into three interconnected parts:

- Model;
- View;
- Controller.

This is done to separate internal representations of information from the ways information is presented to and accepted from the user.

The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.

MVC: Model View Controller

Components:

- The model, is the central component of the pattern. It directly manages the data, logic and rules of the application.
- A view can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible.
- The controller that accepts input (from the view) and converts it to commands for the model.

MVC: Model View Controller

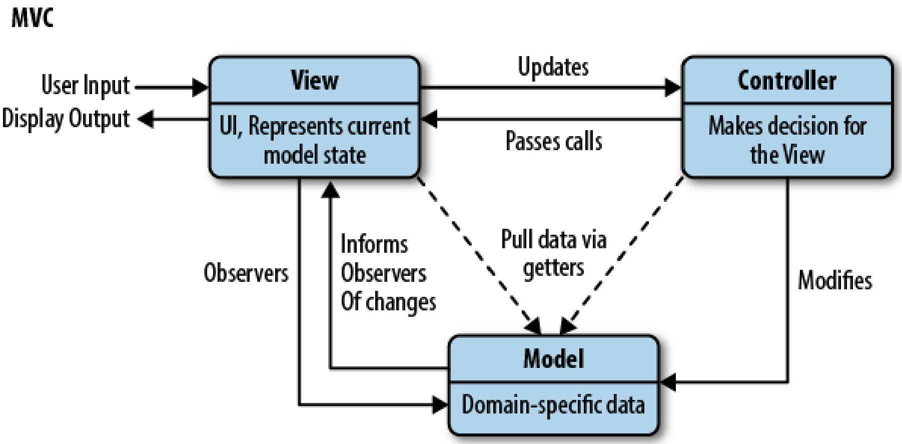
Components:

- The model, is the central component of the pattern. It directly manages the data, logic and rules of the application.
- A view can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible.
- The controller that accepts input (from the view) and converts it to commands for the model.

Interactions:

- The model is responsible for managing the data of the application. It receives user input from the controller.
- The view means presentation of the model in a particular format.
- The controller responds to the user input and performs interactions on the data model objects.

MVC: Model View Controller



MVC: Model View Controller

Advantages:

MVC: Model View Controller

Advantages:

- Simultaneous development, Multiple developers can work simultaneously on the model, controller and views.

MVC: Model View Controller

Advantages:

- Simultaneous development, Multiple developers can work simultaneously on the model, controller and views.
- High cohesion, MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.

MVC: Model View Controller

Advantages:

- Simultaneous development, Multiple developers can work simultaneously on the model, controller and views.
- High cohesion, MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- Low coupling, The very nature of the MVC framework is such that there is low coupling among models, views or controllers.

MVC: Model View Controller

Advantages:

- Simultaneous development, Multiple developers can work simultaneously on the model, controller and views.
- High cohesion, MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- Low coupling, The very nature of the MVC framework is such that there is low coupling among models, views or controllers.
- Ease of modification, Because of the separation of responsibilities, future development or modification is easier

MVC: Model View Controller

Advantages:

- Simultaneous development, Multiple developers can work simultaneously on the model, controller and views.
- High cohesion, MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- Low coupling, The very nature of the MVC framework is such that there is low coupling among models, views or controllers.
- Ease of modification, Because of the separation of responsibilities, future development or modification is easier
- Multiple views for a model, Models can have multiple views

MVC: Model View Controller

Disadvantages:

MVC: Model View Controller

Disadvantages:

- Code navigability, The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.

MVC: Model View Controller

Disadvantages:

- Code navigability, The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.
- Multi-artifact consistency, Decomposing a feature into three artifacts causes scattering. Thus, requiring developers to maintain the consistency of multiple representations at once.

MVC: Model View Controller

Disadvantages:

- Code navigability, The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.
- Multi-artifact consistency, Decomposing a feature into three artifacts causes scattering. Thus, requiring developers to maintain the consistency of multiple representations at once.
- Pronounced learning curve, Knowledge on multiple technologies becomes the norm. Developers using MVC need to be skilled in multiple technologies.

To be continued. . .