Department of Computer Science & Applied Physics

Building a GUI with Model-View-Controller

Model-View-Controller (MVC) was invented by the Norwegian Trygve Reenskaug in 1979 while working in a research team at Xerox PARC (Palo Alto Research Center). PARC also produced a string of revolutionary inventions during this time, including the windows UI, icons, desktop, mouse, laser printing and Ethernet. MVC arose from a requirement to develop a flexible framework capable of facilitating the development of GUI windows and controls. All modern GUI systems, including MS Windows, Android and IOS, are based on MVC. Understanding the how the framework operates will greatly enhance your knowledge of GUI programming and general system design.

At the heart of MVC is a ***separation of concerns*** (a coarse-grained instance of the Single Responsibility Principle) between the three key components of the framework:

- **Model**: The model represents knowledge and may manifest itself as a single object or a complex aggregation of objects.

- **View**: A view is a visualisation of all or part of the model and is intended for human consumption and interaction. It may highlight certain attributes of the model and obscure or hide others. In this sense, a view can be considered as a type of presentation filter. There is a 1:*n* relationship between the model and its views.

- **Controller**: A controller acts as an intermediary between the user of a system and its model. The controller receives requests from a user for CRUD operations and dispatches these requests to the model. Updates are pushed from the model to the view.

MVC increases reusability by decoupling data presentation, data representation and application operations, thus enabling multiple simultaneous data views. It also facilitates maintenance and extensibility through this decoupling of software layers. It is important to understand that the MVC framework does not just apply to GUIs and GUI controls. MVC scales to an architectural level, manifesting itself in the form of the *n*-tier client-server architecture, especially the Model-2 framework.

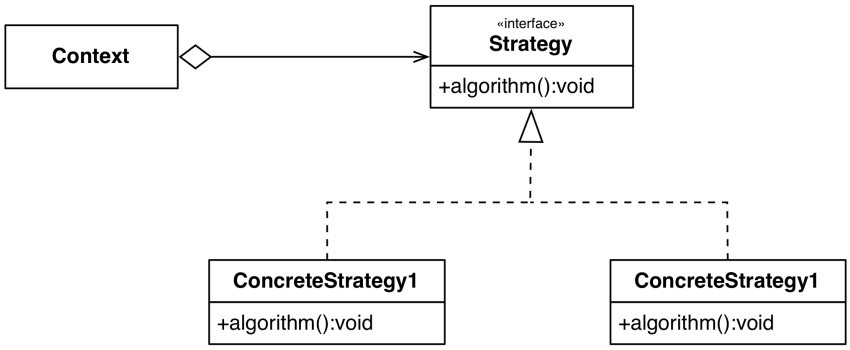
MVC – Three Patterns in One

MVC is ***a composite pattern[1] composed of the interaction of three basic design patterns: strategy, observer and composite***. A strategy pattern is applied to select the model to use. The model itself exploits the observer pattern to notify its different views of a state change. As GUI components, the views themselves are an implementation of the composite pattern, where the set of windows and controls form a tree-like structure. The observer pattern has already been discussed in the practical on *Event Notification and Method Encapsulation*. The strategy and composite patterns are described below.

[1] The term composite pattern has two meanings: it may refer to the interplay of two or more of the GoF design patterns such as MVC or the composite pattern described by the GoF, which uses a tree structure to create whole-part hierarchies. To avoid confusion, the former use of the term will be shown in italics.

The Strategy Pattern – Encapsulate a Family of Algorithms

The intent of the strategy pattern is to enable a class to switch easily between algorithms without any monolithic conditional statements, by defining an abstract policy for performing an algorithm and allowing multiple mechanisms that may comply with that policy. The strategy pattern defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it. Consider the UML diagram below:



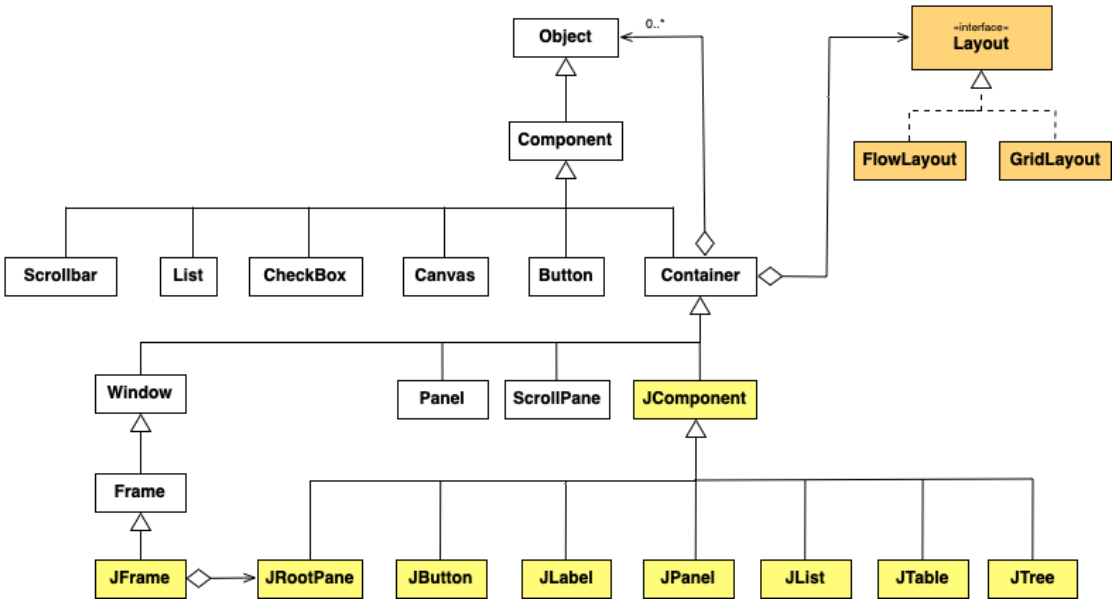
A Context is composed with an instance of the interface Strategy, which declares a method that all concrete subtypes must implement. The implementation however can vary between the different subtypes. Switching the algorithm involves setting the value of the instance variable held by the context to some other instance of Strategy. Note that ***the composition relationship enables this switching to be done dynamically***, enabling the context to change algorithm when an internal state change requires it to do so. This form of switching avoids the use of conditional statements that may be difficult to maintain. The algorithms implemented by subtypes of Strategy can be related (through inheritance) or unrelated (they just implement the same Strategy interface). The interface and strategy methods used by Context must be broad enough to accommodate a disparity of implementations. Note that the decision on the algorithm to use rests with the client. Consequently, the client must know that there are different strategies available for a given context.

In MVC, the strategy pattern is used to select the model. For example, a drop-down list box may use a simple string array as a model. A more complex version may maintain a map that relates strings to objects, which can colour or render more complex visualisations. The strategy pattern is also used in GUI frameworks to enable programmers to apply one of many possible layout managers to a GUI container. For example, the Android GUI framework allows users to switch between layouts such as LinearLayout, TableLayout and RelativeLayout. The subtypes of Layout themselves are composite objects that permit other layouts and controls to be added to them. Similar mechanisms are available in Swing (see the relationship between the class Container and the interface Layout in the Swing inheritance hierarchy below).

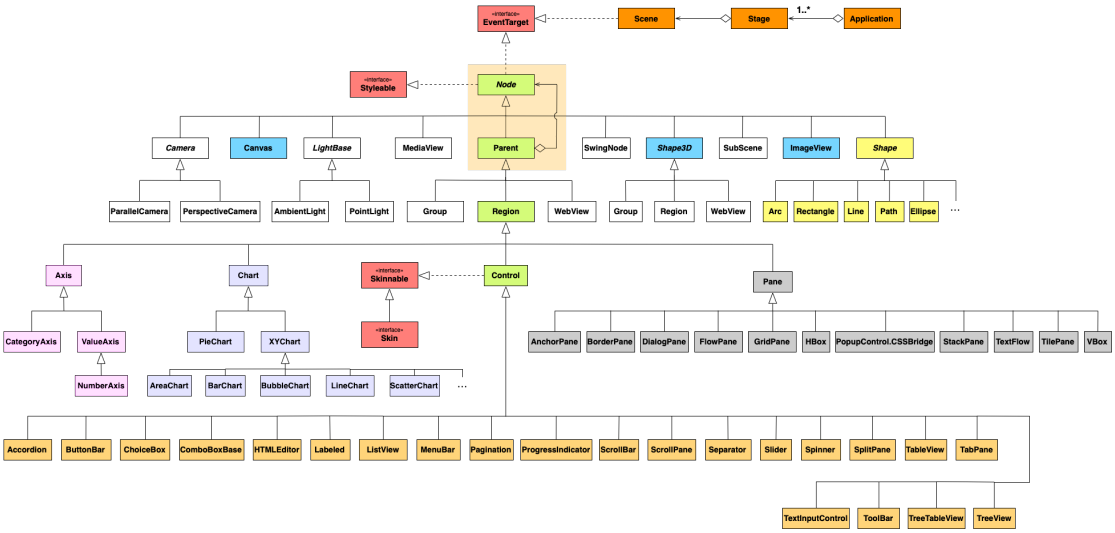
Composite Pattern – Divide and Conquer with Trees

The composite pattern allows the building of object structures in the form of trees that contain both compositions of objects and individual objects as nodes. The intent is to compose objects into tree structures to represent part-whole hierarchies, enabling individual objects and compositions of objects to be treated uniformly. In practice this involves the creation of a container class (a composite node) that uses a collection to store other nodes. Leaf nodes do not have any children and are not required to manage a collection. Both leaf nodes and composite nodes expose the same interface, allowing both classes to be treated uniformly. Consider the

UML diagram of the **Swing** hierarchy below. The class Component defines the leaf nodes in the GUI framework. The classes List, CheckBox and Button are all leaf nodes, as each class, through inheritance, IS-A Component. The class Container **extends** Component (IS-A Component) and is composed with a Component (HAS-A Component). The classes Window, Frame and Panel are all Containers and are all also Components, allowing a Panel to be added to another Panel recursively, i.e. containers support nesting. By starting with a container, e.g. a Frame, components and other containers can be added to create a complex GUI, but the relationship between all the GUI components is really a tree structure.



In the Swing framework, the class Component defines a ***paint(Graphics g)*** method that is overridden by each component and is used to draw its visualisation in a GUI. Each GUI component is a visualisation of a model (simple or complex) used to represent the state of the component. For example, a Button might have states of up or down and will be painted differently for each state. When a state change occurs in the model, the model notifies the view (using an observer) and tells the view to re-paint itself. As the composite pattern yields in a tree structure, a call to ***paint()*** on the top level window or frame can be propagated to every component that comprises the GUI (using a depth-first traversal of the tree).



The **JavaFX** framework is intended to replace **Swing** and also ***uses the composite pattern to create GUIs from an explicit tree of nodes***. JavaFX GUIs are cross-platform and can execute

on Windows, Linux, Mac, iOS, Android and Raspberry Pi platforms. The key abstractions in the JavaFX hierarchy are Node, Parent and Control. An instance of Application can be configured with one or more Stage objects, each of which can have a Scene (a **theatre metaphor**). The class Node is the base class for a Scene graph node. A Scene graph is a **forest**, a set of tree data structures in every item has *0..1* parent and each item is either a leaf node or a composite node, with a single **root tree node** as an ancestor. Each item in the scene graph is an instance of Node.

Branch nodes are instances of Parent, whose concrete subtypes include Group, Region, and Control and their derivations. Parent ***is the base composite class for all nodes that have children*** in the scene graph and handles all hierarchical Scene graph operations, including adding/removing child nodes, checking bounds and executing layouts. The class Parent extends Node (IS-A Node) and is composed with a Node (HAS-A Node). Leaf nodes are classes such as Rectangle, Text, ImageView that cannot have children. The class Region is the base type for all JavaFX node-based UI Controls and all **layout containers**. It is a resizable Parent node and can be styled from a CSS or programmatically. The class Control is the base class for all user interface controls and defines a node in a Scene graph that can be manipulated by the user. Controls support explicit skinning, including the use of CSS, to simplify customising a control's appearance and to separate the MVC components from one another.

Flyweight Pattern – Save Memory by Sharing Object Instances

It is often the case that an application may need to create a large number of objects that have either all the same state or differ only slightly from each other. Consider the *String* class for example. When a new *String* is created, the JVM checks a ***string pool*** to see if that sequence of characters already exists and shares the instance if it does. The *String* class acts as a flyweight; new instances of *String* are only created when they are required, i.e. it is either not in the string pool or the *new* keyword was used against a constructor. For this reason, instances of *String* are immutable, i.e. they cannot be changed after being created. Immutability therefore, provides an ideal mechanism for applying a flyweight, as object state cannot change.

In situations where the difference in state is only slight, the flyweight pattern also provides a solution to limit the number of objects required to be instantiated. Consider the example of a computer game that needs to generate 10,000 oak trees and render them on a screen. If the trees only vary in their position on the screen (x, y and z coordinates), their height and width, then this ***extrinsic state*** can be passed into a single instance of the tree object that can then be asked to paint itself. The internal state of the tree object, e.g. bark colour, leaf colour etc., represents the ***intrinsic object state*** and is shared by all 10,000 oak trees. As the tree object is really just painting itself using the extrinsic state passed in at run time, the it acts like a template or stamp. Flyweights are often combined with the composite pattern to implement shared leaf nodes.

GUI frameworks ***combines MVC with flyweights to reduce the number of objects*** required to paint a component on the screen. Each component can be configured with a flyweight renderer that is responsible for doing the painting. Consider the example of a table GUI component with 10,000 rows of data to display. If painting each row necessitated the instantiation of a separate object to paint the object data, it would result in 20,000 objects being created. Using a flyweight will reduce the number to 10,001, i.e. just one object of overhead to do the painting.

Exercises

In this lab we will use the JavaFX framework to create a simple application that illustrates how MVC enables the creation of custom controls and will then examine where the composite, strategy, observer and flyweight patterns are present in the framework design.

- This lab requires that you have **Java FX** installed on your computer. If you have not done so already, **download and install Java FX** from **openjfx.io**.

- Download the Zip archive titled "*Model-View-Controller Lab (Finished Source Code)*" on Moodle and extract the packaged files into a new Eclipse project called **MVCSimpsons**.

- Add the correct JAR files from the Java FX lib directory to the ***module-path*** of the **MVCSimpsons** Eclipse project.

- Examine the module descriptor ***module-info.class***. Notice that we are consuming the *javafx.controls*, *javafx.base and javafx.graphics* modules and that the module *javafx.graphics* has is required through a **transitive dependency** by one or both of the other modules.

- Run the application and examine the output, study the source code and then *identify which parts of the application play the roles* of a *Model*, a *View* and a *Controller*.

- Identify where the **composite**, **strategy** and **observer** patterns are present in the implementation and how they interact with each other in the MCV framework. *What role can a flyweight play in a GUI application and in what context could a flyweight be used in this application?*

- The following code snippet draws a hexagon shape using JavaFX. Use the code to *create a custom GUI control that responds to click gestures from a user* and can be added to a GUI as a component.

```
Polygon p = new Polygon();
for (int i = 0; i < 6; i++){
  p.getPoints().addAll(new Double[]{
    (100 + 50 * Math.cos(i * 2 * Math.PI / 6)),
    (100 + 50 * Math.sin(i * 2 * Math.PI / 6))
  });
}
```