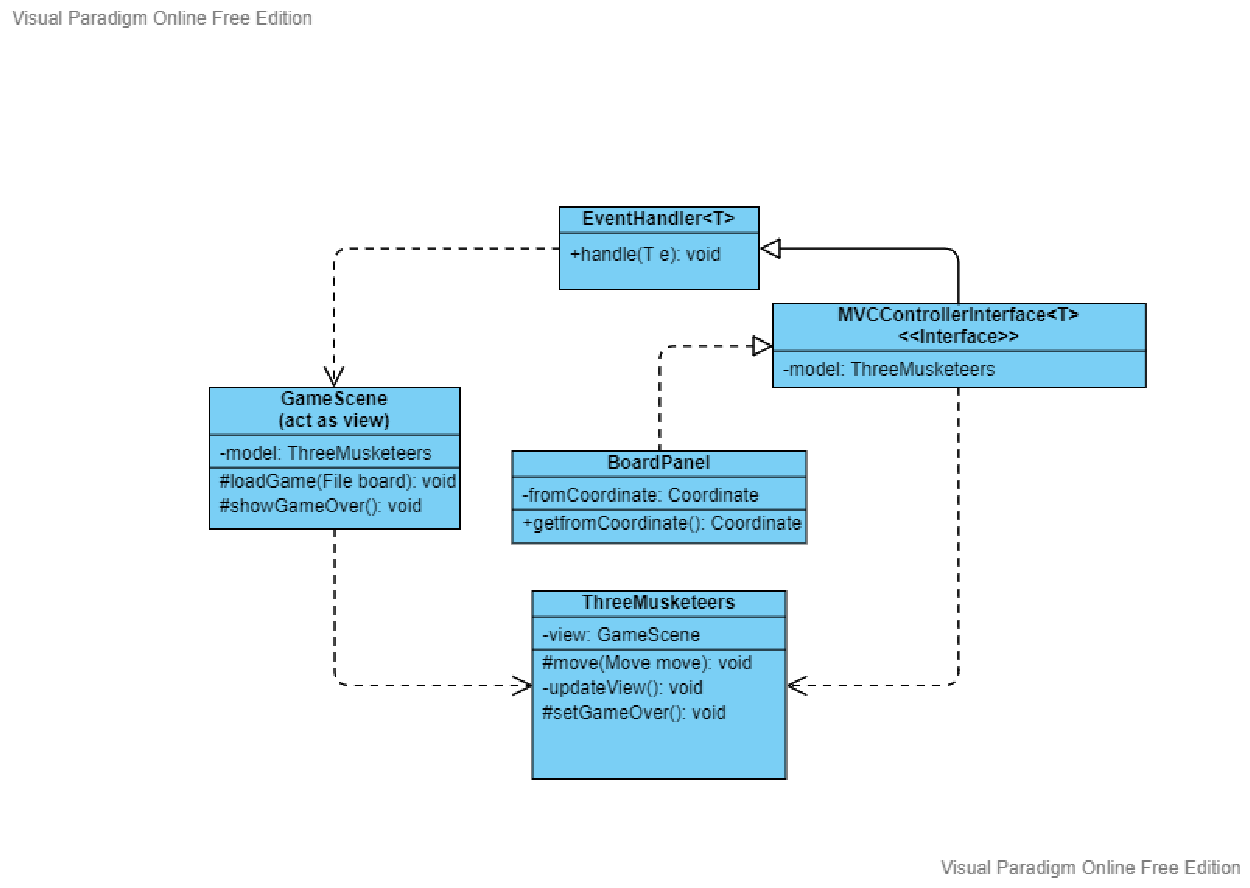
# 3Musk4Dev
# Design Patterns

# Table of Contents

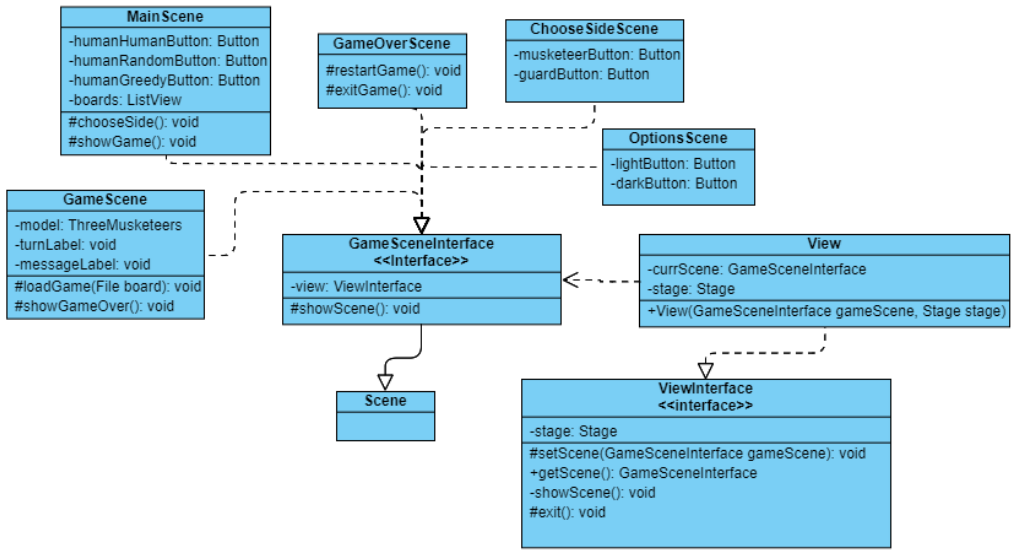
## MVC (Observer, Strategy & Composite): GUI



**Description and solution:**
GameScene.java in our project acts as the view component for MVC, where the user can directly input their commands. Boardpanel.java is the controller that interprets the user's command into machine-friendly codes, and ThreeMusketeers.java is the model that contains the core.
**Functionality:**
A feedback loop is created among the three concrete classes GameScene, BoardPanel and ThreeMusketeers. The human presses the buttons and keys, and the BoardPanel is triggered. BoardPanel then calls handle() to update its fromCoordinate, and to update ThreeMusketeers accordingly. The ThreeMusketeers will then refresh GameScene according to its current state.

By dividing the feedback loop into subtasks, the flow of commands and changes are more consistent, and can be more easily tracked. This will make bug tracing easier for developers. This structure can also serve as a basis for other software development.

## Strategy: View



**Description and solution:**
View is the scene manager class. It contains scenes which will change over time. The strategy pattern helps to group scenes together and changes the scenes with few lines of code.

**Functionality:**
When launching the game, a View object is created, and the MainScene (i.e. the class representing the main menu) is attributed to currScene.

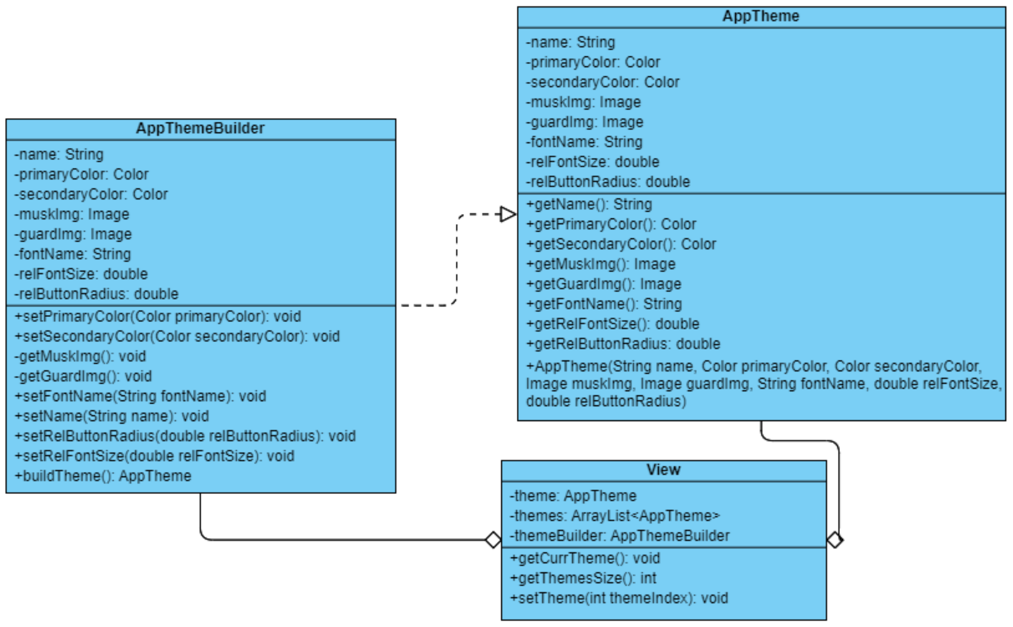
Each GameSceneInterface has to construct its own scene in its showScene() method. This method will be called by the showScene() in ViewInterface. In the concrete implementation of GameSceneInterface classes, attributes and methods are freely added into each scene for its purposes. The UML diagram above shows some of the more important methods used in each scene.

View has the property to change scenes, and asks the GameSceneInterface to show their scene on the stage, as well as to exit the game.

Since scenes may need to be changed often during a gameplay, separating and abstracting scene classes not only makes the View class look tidier, but it could also facilitate the development of new scenes (e.g. a game over screen, or an settings screen), as well as to regulate the flow of existing scenes (e.g. after the game over screen, a side picking scene should not be followed directly.

A similar structure is applied to View switching themes.

## Builder: Themes

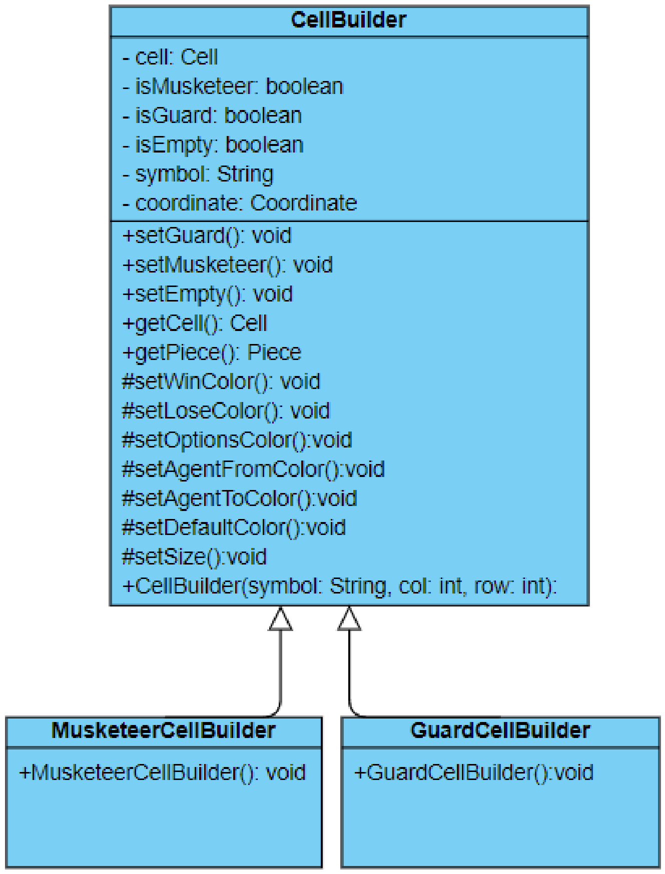

AppTheme
-name: String
-primaryColor: Color
-secondaryColor: Color
-muskImg: Image
-guardImg: Image
-fontName: String
-relFontSize: double
-relButtonRadius: double
+getName(): String
+getPrimaryColor(): Color
+getSecondaryColor(): Color
+getMuskImg(): Image
+getGuardImg(): Image
+getFontName(): String
+getRelFontSize(): double
+getRelButtonRadius: double
+AppTheme(String name, Color primaryColor, Color secondaryColor, Image muskImg, Image guardImg, String fontName, double relFontSize, double relButtonRadius)
AppThemeBuilder
-name: String
-primaryColor: Color
-secondaryColor: Color
-muskImg: Image
-guardImg: Image
-fontName: String
-relFontSize: double
-relButtonRadius: double
+setPrimaryColor(Color primaryColor): void
+setSecondaryColor(Color secondaryColor): void
-getMuskImg(): void
-getGuardImg(): void
+setFontName(String fontName): void
+setName(String name): void
+setRelButtonRadius(double relButtonRadius): void
+setRelFontSize(double relFontSize): void
+buildTheme(): AppTheme
View
-theme: AppTheme
-themes: ArrayList<AppTheme>
-themeBuilder: AppThemeBuilder
+getCurrTheme(): void
+getThemesSize(): int
+setTheme(int themeIndex): void

**Description and solution:**
Themes are objects that carry multiple properties, like primary color, secondary color, font size and name, etc.  Creating an instance by passing these parameters through a constructor is tedious. By using a builder, the set-up of the parameters can be separated from the construction, enhancing the flexibility of our code, as well as regulating method access protection.

**Functionality:**
An AppThemeBuilder object contains the same set of parameters as the AppTheme object. The builder will have a set of default values, and developers can alter the parameters by using setters in the builder class. With the buildTheme() method, an AppTheme instance will be returned, set with the current parameter values in the builder. The AppTheme instance will then be used in View and its values are fixed, since there are no setters in AppTheme objects. This prevents developers from accidentally altering the “finalized values” in AppTheme objects.

## Builder: Cells

**Description and solution:**
In order to separate the board's cells, the builder design pattern can ease the process of initializing new cells when the board is first built. This will allow for further customizability for cell options in the future for developing more variations of game modes.



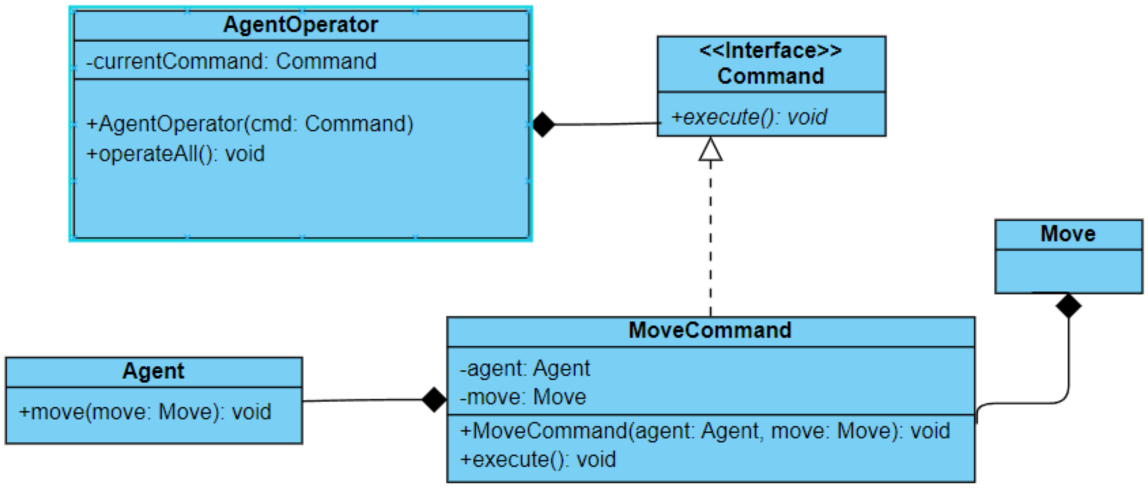
**Functionality:**
When loading the board, CellBuilder will read the string representation of the current cell and immediately build the cell accordingly. This will allow for more customizability of the cells and aggregates the need of calling multiple constructors to build a cell.

All of the composites of a Cell, namely Piece Type (Musketeer, Guard) and Coordinate will be set instantaneously with less complex constructors.

This improves the overall code quality by reducing the need to collect arguments and create new instances.

By passing three parameters with the loadBoard function in the Board class, namely the symbol of the piece and the coordinates, col and row, CellBuilder can automatically construct the pieces with its supposed features.

## Command: Moves

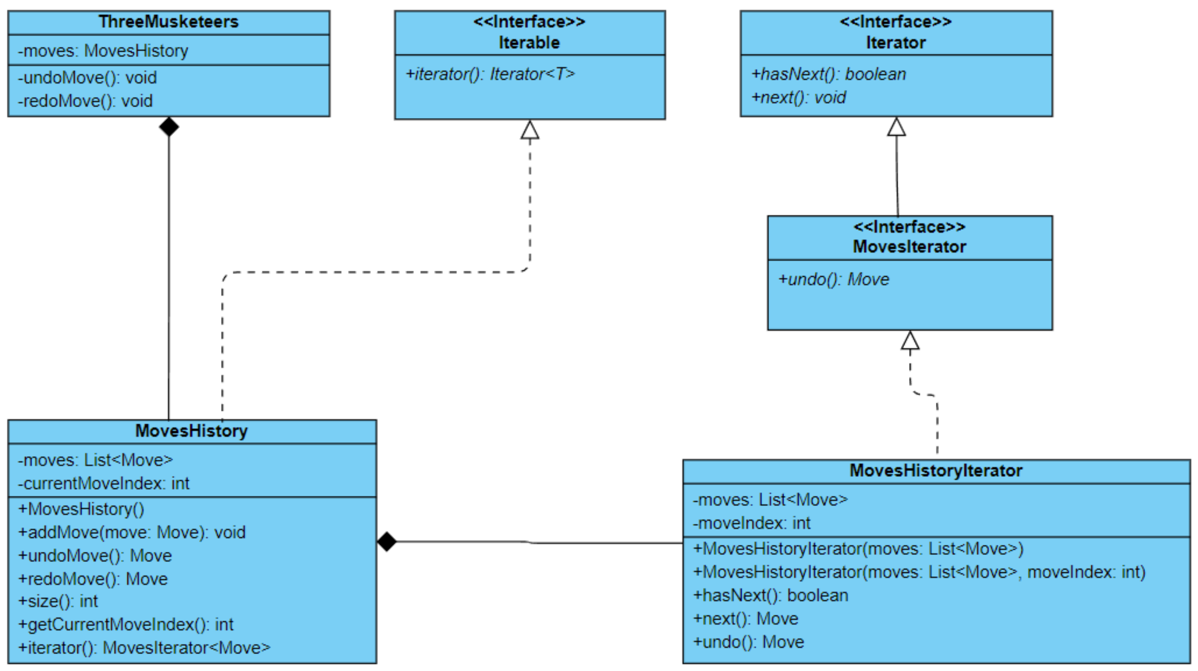


**Description and Functionality:**
Moves made in the game have relatively complex implementations, which a commander class like Agent need not to know. Through utilizing the command pattern, the moves can be easily separated into its own classes, and the game could technically possibly be extended by allowing new types of moves in the move handler classes.

**Functionality**

When an agent moves, the AgentOperator class will act as the commander to enable a movement command MoveCommand so that the Agent will execute its move(Move move) method.

## Iterator Pattern: Undo & Redo Functionality



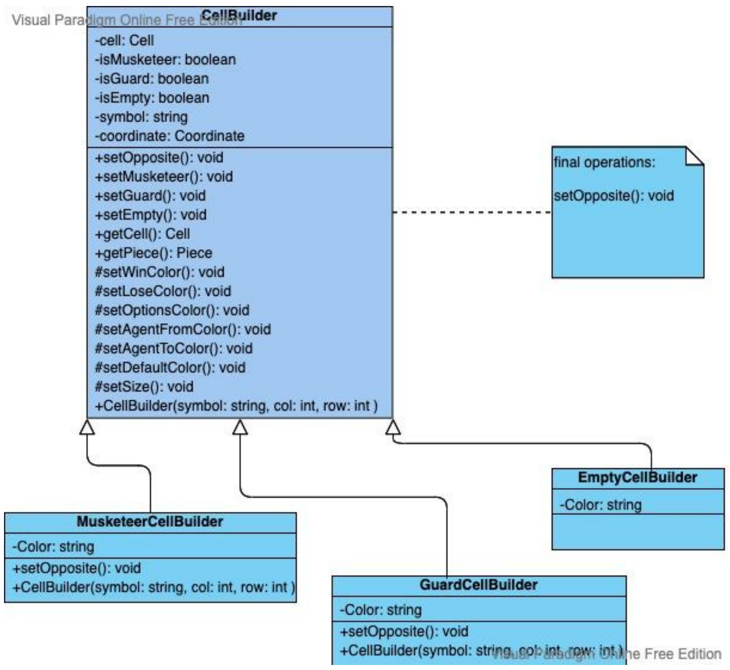
**Description & Solution:**
The iterator pattern implemented onto the undo function of the board's moves will allow players to not only undo, but also redo their moves through the MovesIterable.

**Functionality:**
The Board contains moves, which stores the moves history in the MovesHistory class. MovesHistory implements MovesIterable and can add new moves to the iterable list of class MovesIterator.

In MovesHistory, addMove(Move move) adds a move to the list of moves, and increments the current moveIndex by 1, to store how many moves have been moved throughout the game.

Upon redoing moves and playing an alternate move, nextMove(move) will remove all previous moves that followed the current move, and points the next move to the move that is passed as an argument.

## Template method: CellBuilder

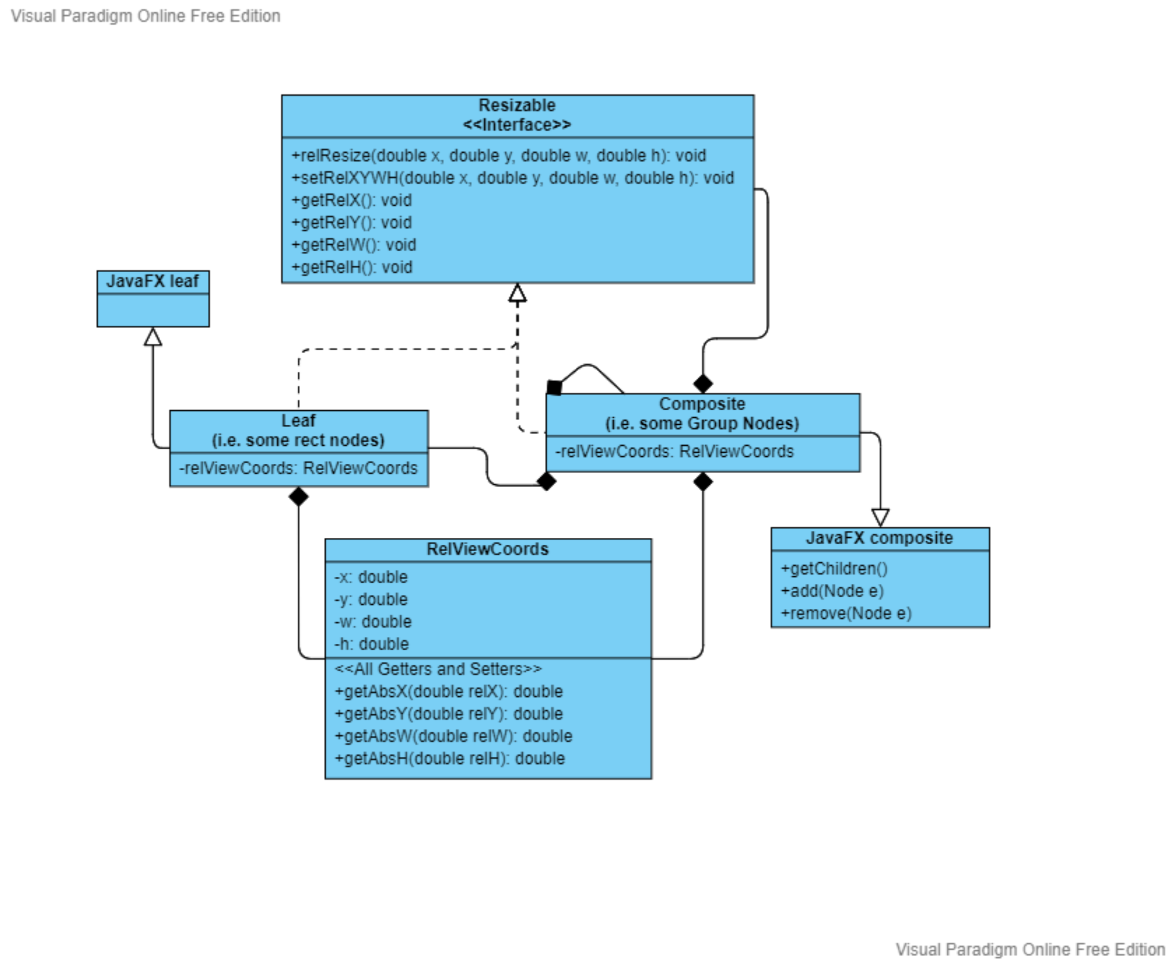


**Description & Solution:**
The cell builder that we are going to implement has another design pattern known as the template method, which contains abstract and concrete classes. The CellBuilder is the abstract class that provides the blueprint and it has three different subclasses that build different types.

**Functionality:**
Similar to the functionality described above in the builder section, the CellBuilder class is designed to construct cells in a more concise way. The three different sub-builder classes are made so that each can handle different build tasks.

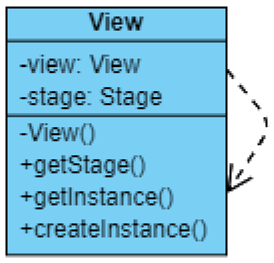
This can ensure that all the classes will have all the necessary functionality the cells need and can make code more concise.

## Composite Pattern: Resizable functionality


Visual Paradigm Online Free Edition
Resizable
<<Interface>>
+relResize(double x, double y, double w, double h): void
+setRelXYWH(double x, double y, double w, double h): void
+getRelX(): void
+getRelY(): void
+getRelW(): void
+getRelH(): void
JavaFX leaf
Leaf
(i.e. some rect nodes)
-relViewCoords: RelViewCoords
Composite
(i.e. some Group Nodes)
-relViewCoords: RelViewCoords
JavaFX composite
+getChildren()
+add(Node e)
+remove(Node e)
RelViewCoords
-x: double
-y: double
-w: double
-h: double
<<All Getters and Setters>>
+getAbsX(double relX): double
+getAbsY(double relY): double
+getAbsW(double relW): double
+getAbsH(double relH): double
Visual Paradigm Online Free Edition

**Description and solution:**
To make the JavaFX components resizable, one must realize that both the groups (composites) and the leaves of the scene tree must be all resizable. Therefore, by having a composite pattern, when the scene is resized, the root calls its components to resize, and each composite calls its children to resize. As a result, the whole scene tree could be resized.
**Functionality:**
Any resizable will have their relative view coordinates (RelViewCoords) to their direct parent. The “setters” and “getters” of the interface obtain the information from the RelViewCoord class. Using the information that RelViewCoords provides, as the screen is resized, each and every component can be resized properly by passing the relative xywh values **to the whole screen, not the parent** to relResize() function, where the leaves will eventually be resized to the correct size and position relative to the screen.

## Singleton Pattern: View



**Description and solution:**
View is the overall stage manager class, and it is on top of every other class (except Main). Therefore, there should only be one instance of it, to prevent unnecessary memory loss. It is best to regulate the creation of instances of View so as to make sure there is only one of it.

**Functionality:**
The View constructor is hidden from all other classes, so other classes cannot create View instances through the constructor. Instead, they only create it through the createInstance method, which is made to be static. This method will only create an instance when there is none. The new instance will be stored in View.view, and the stage instance will be stored in View.stage, where both are made to be privately static. This way, developers or clients cannot create multiple instances of View objects.