# Software Architecture

## System Architecture

For the development of the System and its graphical user interface, the Model-View-Controller (MVC) pattern will be used. This type of pattern is widely adopted in the development of graphical interfaces because the code is divided into three components: Model, View, and Controller. This approach makes the code much more readable and consequently increases the maintainability of the system.

As mentioned earlier, the code is divided into three blocks, each of which contains a part of the code that contributes to implementing the functionalities of a graphical interface.  
Here are the blocks and how they are used:

* **Model**: This block contains the data used in the system. In our specific case, the model implements the various functionalities of the system.
* **Controller**: This is the core of the MVC pattern. It implements the logic of the graphical interface. For example, the action that occurs after pressing a button is implemented in this block.
* **View**: This block is used to create the actual graphical interface. It defines where to place buttons, labels, and also the interface’s font.

Its functioning can be seen in the displayed figure: a user—who in our case is using the application—sends a request to a controller. The controller requests and retrieves data from the model, processes it, and returns it to the view, which displays the response to the user.

Each block communicates with the others: the controller manages the buttons defined in the view; the view displays data stored in the model block. This pattern maintains separation between components, which results in greater flexibility and code reusability. This way, I can create views for different devices where I want to integrate the application, without modifying the model or the controller.

In our case, the application will follow the MVC pattern using JavaFX as the standard programming language for building the graphical user interface. This choice promotes consistency and ease of development, as JavaFX is widely used for creating user interfaces in Java desktop applications.

## Sprint 1

### Class Diagram and Pattern used

A screenshot of a computer

AI-generated content may be incorrect.

For the first sprint, we have defined the class diagram based on the planned features. By employing a Model-View-Controller (MVC) architecture, we can divide the diagram into three main sections: Model, View, and Controller.

The Model encompasses the classes responsible for managing the system's data and business logic. The cornerstone class is CanvasModel, which implements the CanvasModelInterface. This approach promotes low coupling and abstracts the concrete implementation of CanvasModel. Within it, the main methods allow for the manipulation of the LinkedHashMap containing ShapeData instances (the data abstraction of the figures displayed in the View). These manipulations occur through the Command Pattern: each operation is encapsulated in a command object, allowing the client requesting the operation to be unaware of its implementation details. The CommandManager acts as the "Invoker" in this pattern, orchestrating the execution of commands. The ShapeData class is a crucial abstraction, representing the data of the figures. It is an abstract class extended by specializations such as LineData, RectangleData, and EllipseData. The creation of these figures is managed by the Factory Method Pattern (or more accurately, an Abstract Factory if ShapeDataFactory had multiple methods, or a set of Concrete Factories implementing a common Factory interface, as in your case), which handles the instantiation of new figures when requested by the Controller. Finally, the FileManager class, by implementing FileManagerInterface, manages the saving and loading of .pr files. The widespread use of interfaces that are then implemented by concrete classes is a practice that significantly improves decoupling within the system.

The View consists of standard JavaFX classes like Pane and Shape, augmented by the CanvasView class which implements CanvasViewInterface. This custom class handles specific operations to be performed on the view, such as adding or removing the graphical representations of figures.

The Controller acts as an intermediary between the Model and the View. It manages user interface events (like mouse clicks or button presses) and, when operations that modify the system's state are desired, it invokes the appropriate commands that act on the Model. Thanks to the Observer Pattern, the Controller is notified of changes occurring in the Model (being a ModelObserver itself), thus allowing it to update the View efficiently and maintain visual consistency.

## Sprint 2

### Class Diagram and Pattern used



In the Model layer, the management of shape data and its manipulation has been further developed. A key addition is the ShapesClipboard class, which now centralizes the logic for copy-paste operations. This dedicated component manages a collection of copied ShapeData instances, including an offset to control their placement upon pasting, thereby streamlining how clipboard functionality is handled within the CanvasModel. The command pattern, previously established for operations, has been considerably extended. New command objects, all implementing the common Command interface, now encapsulate a wider array of actions. These include commands for managing the Z-ordering of shapes, such as BringToFrontCommand and SendToBackCommand, which meticulously record the previousState of the shapes to ensure accurate undo functionality. Similarly, clipboard actions are now managed by CopyShapeCommand, CutShapeCommand, and PasteShapeCommand, with the latter two also preserving the previousState. Shape manipulation is further enhanced with DeleteShapeCommand for removing shapes, DeselectAllShapeCommand and SelectShapeCommand for managing selection states, and a suite of editing commands like EditShapeStrokeWidthCommand, EditShapesFillColourCommand, and EditShapesStrokeColourCommand for modifying visual properties. Finally, MoveShapesCommand and ResizeShapeCommand offer granular control over shape positioning and dimensions, respectively. Each of these commands interacts directly with the CanvasModel as its designated receiver, and their consistent inclusion of previousState tracking significantly broadens the scope of undoable operations, leading to a more robust and forgiving user experience.

The View layer has also seen improvements aimed at providing more intuitive visual feedback during interactions. The CanvasView is now equipped with a resizeHandle, typically a Rectangle, and a previewShape. These elements offer immediate visual cues to the user during complex operations like drawing a new shape or resizing an existing one. Supporting methods such as paintPreview, updatePreviewShapeGeometry, and updateResizeHandle are responsible for rendering and updating these visual aids dynamically. Furthermore, a new Highlighter class has been introduced to enhance the visual distinction of selected shapes. This component can manage various highlighting effects, potentially including animations, by keeping track of selectedShapes and any activeAnimatedHighlights, thus making the selection state clearer to the user.

The most substantial evolution has occurred within the Controller and its interaction management, primarily through the formal introduction of a State Pattern. The Controller now adeptly manages different modes of user interaction—such as idling, drawing, selecting, moving, or resizing—by maintaining a currentState which is an instance of the State interface. This architectural shift allows the Controller's response to user input (like mouse events) to vary dynamically based on the active state. Concrete state classes such as IdleState, PaintingState, SelectionState, MovingState, and ResizingState implement the State interface, each providing specific logic for handling mouse events and managing transitions between states. For instance, when the PaintingState is active, mouse drags will result in the creation of an AddShapeCommand, while in MovingState, similar actions will lead to the instantiation of a MoveShapesCommand upon completion. The Controller is responsible for creating instances of these state objects and also for instantiating the relevant command objects based on user actions within the context of the current state. To facilitate the rendering of shapes, a ShapeConverter class has been integrated. This utility, along with its associated ShapeAdapterInterface implementations (EllipseAdapter, LineAdapter, RectangleAdapter), provides a structured way to convert abstract ShapeData from the Model into concrete JavaFX Shape objects suitable for display in the CanvasView. Lastly, a SelectionPropertyObserver has been introduced as a specialized ModelObserver. This observer monitors the CanvasModel specifically for changes in shape selection and updates selectionBoundNodes—UI elements like buttons or menu items—accordingly, ensuring that UI controls are appropriately enabled or disabled based on the current selection context.