Kanvas Design Document

SWE4403

Instructor: Julian Cardenas

Project Team #: 3

Names: Justen Di Ruscio, Ethan Garnier

Respective Student Numbers: 3624673, 3658981

Due Date: March 21st, 2022

Contributions:

Contributions by both team members are completely equal in all aspects of the project, including programming, design, report, and developing the video. Kanvas was a completely collaborative effort. The use of software design patterns, adhering to strong design principles, and the architecture of the software allowed for an easy, fluid partnership. The requirements, design, and implementation of Kanvas evolved throughout the project, yet all these adjustments were carried out as a team. We would be happy to show previous UML diagrams, discussions, and implementations to depict this collaboration. Through this project, we not only learned how software design patterns can improve software quality, but also learned how they can improve teamwork.

Group Member	Signature
Justen Di Ruscio	
Ethan Garnier	

Table of Contents

1.0 Introduction

Kanvas is a minimal Microsoft Paint like utility that provides users with the ability to easily draw lines and shapes of different colors and sizes. The application presents users with a blank white canvas and provides them with various drawing tools that can be used to add detail to their canvas.

Kanvas was written entirely in Kotlin, an object-oriented programming language developed by JetBrains. The decision to develop this application in Kotlin was made for the modern object-oriented features that the language presents. The Kanvas application has been developed as a multiplatform desktop application thanks to the Compose Multiplatform framework, a brand-new desktop and web UI framework also developed by Jet Brains.

The goal with developing Kanvas is to demonstrate an easy alternative to desktop application development and to depict the application of object-oriented design patterns to create easily maintainable and extendable software. Depicted in figure 1 is a top-level view of Kanvas' class hierarchies.
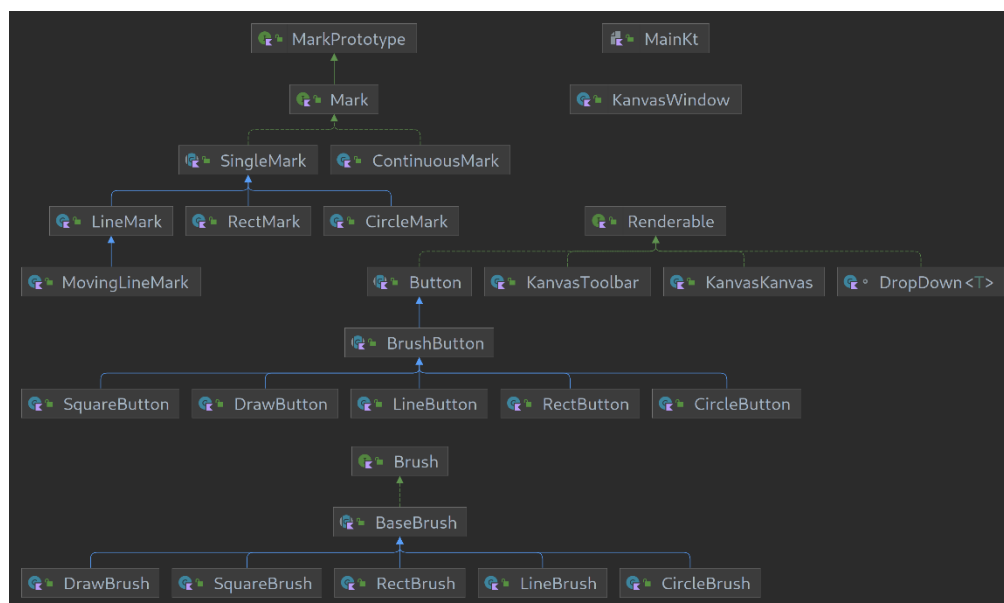

Figure 1 – Class Overview

2.0 Program Overview

Shown in figure 2 is a sample of the Kanvas program. It's composed of two main sections: the toolbar and the drawing canvas. From the toolbar, the user can select from an assortment of *brushes* via the Icon Buttons at the right side of the toolbar. For any of the brushes, the program provides the ability to adjust both the current brush size and brush color from the respective drop-down buttons on the toolbar. As the user interacts with the drawing canvas, which is referred to as applying *strokes* with their selected brush, *marks* are placed upon the drawing canvas, which are associated to their selected brush.

Each brush can be *stroked* along the drawing canvas to apply a *mark* to the canvas. However, depending on the selected brush, and therefore the mark that will be applied to the canvas, the application of the stroke changes. Consider the drawing brush, shown at the far left of figure 1. When a user has their mouse button clicked (places the brush on the canvas) and they move their mouse (the brush), the mark on the canvas grows until they release the button. On the other hand, for one of the shape brushes, like the circle, square, and rectangle marks shown on the right of figure 2, as the use moves their mouse with the mouse button depressed, the entire mark itself moves. In summary, the application of strokes on the canvas change to correspond to the most natural action for the currently selected brush. Finally, a secondary mouse-button click will clear the canvas to prepare for a fresh drawing.
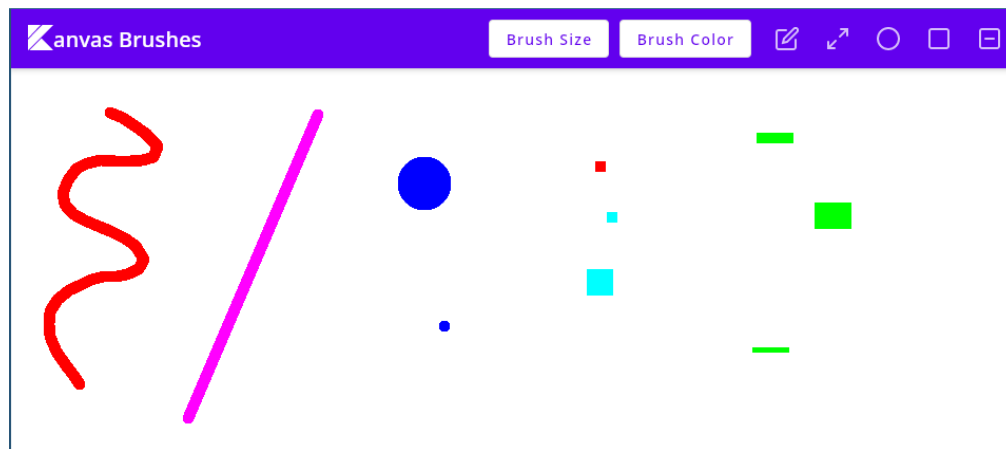

Figure 2 - Kanvas Sample

3.0 Design Patterns

In total, four explicit design patterns were employed in the development of Kanvas. Each of these patterns was chosen to improve the extensibility of Kanvas and to simplify its overall operation. As a note, each design pattern is accompanied by a UML diagram. Since the overall UML diagram would be much too complex for a single figure, only a subset of the entire software's UML diagram that pertains to the specific design pattern is shown in each of these UML diagrams.
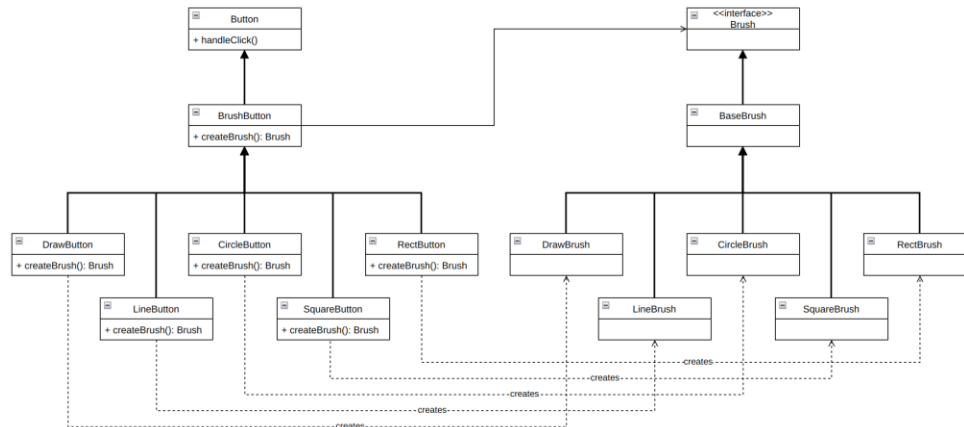
3.1 Factory Pattern

Figure 3 – UML Diagram of Kanvas's Factory Pattern

3.1.1 Reasoning

The Factory Pattern allows for object instantiation to be deferred to different subclasses. In the context of our software, where the respective brushes must be instantiated based on which UI elements the user interacts with, the Factory Pattern is a clear solution.

3.1.2 Application

In Kanvas, the Factory Pattern is applied between the interaction of buttons and the brushes they create. Each of our button UI elements that are responsible for allowing the user to change their drawing tool, which we call brushes, directly inherit from a *BrushButton* abstract class. This abstract class enforces the *createBrush()* abstract method, which returns an object of the *Brush* interface. The *Brush* interface is an interface that all drawing tools within Kanvas inherit from. It provides methods for configuring various attributes of a drawing tool (I.e., color, size, stroke), and methods for applying *marks* to the canvas. Consequently, all button UI elements that inherit from the *BrushButton* abstract class must override the *createBrush()* method in order to create an appropriate brush.

With the Factory Pattern, based on which *BrushButton* is clicked, a different drawing brush will be returned. For example, when the *CircleButton* class has its createBrush() method called as it's pressed, a *CircleBrush* is returned. This delegates responsibility to the *BrushButton* subclasses to return the proper Brush subclass.

From this, we can identify the components of the Factory Pattern. Here, The *BaseBrush* class is the *Product*, while its children, like the *CircleBrush,* are the *Concrete Products*. For the factories, the *BrushButton* is the *Creator* while its children, like the *CirlceButton,* are the *Concrete Creators* in this pattern.

3.1.3 Consequences

As a result of implementing the Factory Pattern for brush instantiation, the ability to add new brushes into Kanvas has been drastically simplified. No existing code must be modified – following the open-close principle. All that is required is a new *BrushButton* and a *Brush* for that button to create. This makes our software extremely customizable for anyone that wants to add

their own drawing tools. Furthermore, each class has a single responsibility, so the code is easily understandable and maintainable.
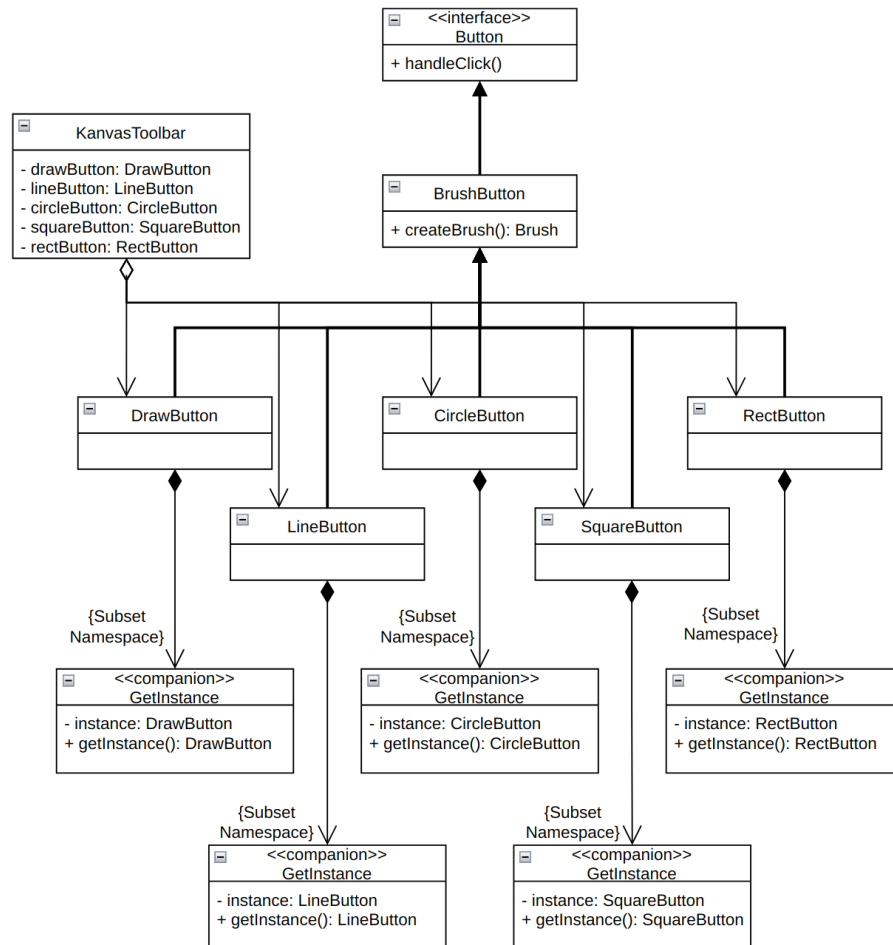
3.2 Singleton Pattern

Figure 4 - UML Diagram of Kanvas's Singleton pattern

Kotlin doesn't have the concept of the *static* lifetime specifier. Instead, to achieve the abilities that *static* provides, *companion objects* must be used. They are effectively nested classes that will automatically be instantiated once for their enclosing class. Nevertheless, the members of the companion objects act identically as they would if they were simply declared static within the enclosing class.

3.2.1 Reasoning
The Singleton Pattern ensures a class has only one instance. In the context of our software, where we have UI elements that exist in only one spot throughout the programs entire lifecycle and perform only a single task, implementing these UI elements as singletons is the most intuitive decision and conveys their usage to future developers.

3.2.2 Application
Each of our button UI elements that are responsible for providing the user with a unique drawing tool serves only one purpose. These elements do not change their operation over the lifespan of

our program, and they remain visible and in the same place the entire time the user is drawing with Kanvas. Thus, each button UI element has had its constructor set as private and has been given a static instance member and a static *getInstance()* method to return the instance member. The static instance member represents the sole instance of the button. Each time *getInstance()* is called after that, the method will return the exact same instance as before. No further instances can be created as the constructor has been set to private. Implementation of what is described above was achieved with Kotlin's companion objects feature. Companion objects allow for static member variables and methods to be created for classes.

3.2.3 Consequences

As a result of implementing the Singleton Pattern for button UI elements, we now have control over the number of UI elements our program creates, and this ensures no memory or computation power is being wasted. Furthermore, the code accurately represents the application itself, as there should only ever be a single button for any one purpose. Should any future modifications be made to these buttons, since the singleton pattern is employed, there is only one spot in the code that requires updates, allowing fast and concise improvements to the program. Beyond this, if future extensions require access to the button, they can easily share the single instance with the use of the *getInstance()* method.
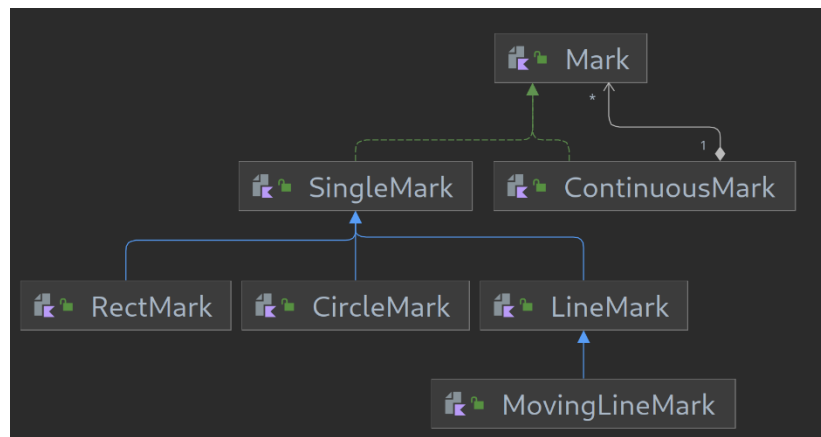
3.3 Composite pattern

Figure 5 – UML Diagram of Kanvas's Composite Pattern

3.3.1 Reasoning

The Composite Patterns allows for objects to be composed into tree-like structures to represent part-whole hierarchies. In the context of our software, strokes on the canvas can be represented by a single mark or a composition of multiple single marks, in the case of the drawing tool. This presents the perfect opportunity for us to implement the Composite Pattern such that we can represent individual marks and nested compositions of marks in the same manner.

3.3.2 Application

There are two types of marks that can be made on the canvas. One such mark is a *SingleMark* object; this represents an individual shape being drawn onto the canvas, and another is a *ContinuousMark*. The *Mark* is itself an interface containing the necessary functions to draw on

the canvas, which both Single and Continuous Marks adhere to. The *SingleMark* children are classes that implement this interface to represent uniquely shaped marks to be drawn on the canvas. Examples of these are the CircleMark class for drawing circles or the RectMark class for drawing rectangles. On the other hand, the *ContinuousMark* class contains a collection of other *SingleMark* objects, that when combined, represent a continuous stroke on the canvas. Thus, the *ContinuousMark* object is our composite object and the singular Mark objects are the parts that compose the *ContinuousMark.*

Identifying the components of the *Composite Pattern* in this diagram is as follows: the *Mark* is the *component interface,* while the *SingleMarks* are the *leaves.* Then, the *ContinuousMark* is the c*ontainer* of this pattern, containing other components.

3.3.3 Consequences
As a result of implementing the Composite Pattern for marks, we can now treat the individual marks and continuous strokes made on the canvas in the same manner. Not only this, but thanks to the composite pattern, much more complex marks on the page could be created when more deeply nested trees are used in future iterations of this software. For example, this could be useful for animations, or marks with many features composed of other marks. This allows for much easier handling of the marks system in Kanvas as we can completely ignore the fact that the ContinuousMark and SingleMark classes have different contents, as they all implement the Mark interface.

For future extensibility the implementation of this pattern allows the *Momento* pattern to be employed. Consider undo-ing and redo-ing marks. If continuous marks were simply a scattering of individual marks, and there was no container class, an undo would remove one of the individual marks that compose the entire stroke. Instead, the undo should remove all the marks that compose that last stroke, which the container class, the *ContinuousMark*, allows.
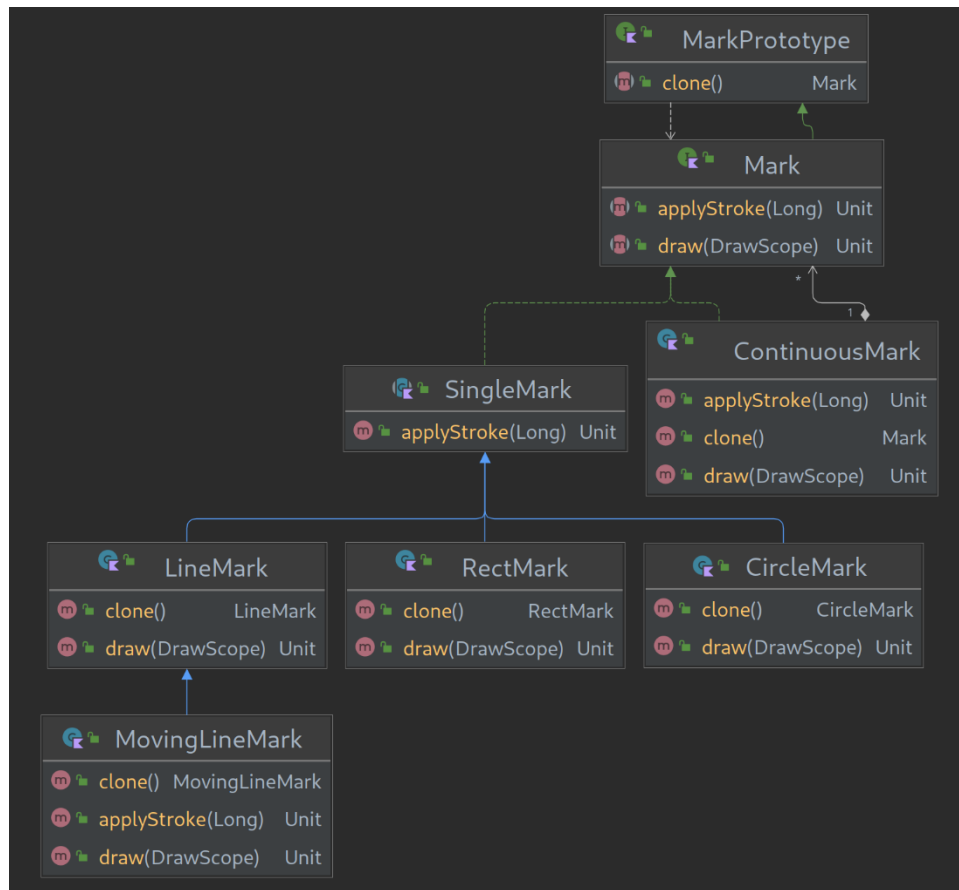
3.4 Prototype Pattern

Figure 6 – UML Diagram of Kanvas's Prototype Pattern

3.4.1 Reasoning

The Prototype Pattern allows for a prototype object to be cloned when needed. This is especially useful when you have an object that can be configured with certain properties, and you need many of these objects with a particular set of properties to be created. In the context of our software, users can configure the color, shape, and brush of the marks being made on the canvas. This situation provides the perfect opportunity to implement the Prototype Pattern, as instead of recreating marks of identical properties, we can simply clone a prototype mark with the desired properties the user currently has configured. This pattern is also essential to the drawing functionality of the tool, as discussed in section 3.4.2.

3.4.2 Application

As the *ContinuousMark* applies its stroke, it clones the previous mark in its collection, applies the stroke to that specifically, and then appends it to its collection. In this case, the *ContinuousMark* can't manually create marks without the *clone()* method, as it's unaware of the type of mark it contains, or what constructor parameters it accepts. The *clone()* method is necessary such that the *ContinuousMark* can continuously recreate marks.

The prototype pattern was implemented by first creating a *prototype interface*, the *MarkPrototype* class. Then, all *Marks* implement the *clone()* method to become *concrete prototype* classes.

3.4.3 Consequences

As a result of implementing the Prototype Pattern for Marks, a consistent interface is provided to recreate all marks, regardless of their type. No matter the constructor parameters they accept, they can easily be copied, with the use of the *MarkPrototype* interface. In the context of Kanvas, this allows the drawing feature to operate.

Another benefit to this interface is that if any future extensions are made to the software, they can uniformly recreate *Marks,* even if they are aware of the type and the necessary constructor parameters. This removes repeated initialisation code and allows complex *Marks* to be created easily.

4.0 Using the Software

Kanvas is built upon the Java virtual machine, so it comes with the flexibility to run anywhere that has the Java runtime installed. A single Java Archive (JAR) is the only distributed artifact for Kanvas, which includes all its dependencies. To run, download the latest Kanvas release, then simply run downloaded Kanvas JAR with Java-11 or greater.

To build the software from source code, the process is greatly simplified with the Intelli-J IDEA IDE. Open Intelli-J and chose to get a project via Version Control (VCS). In the window that pops up, paste the HTTPS clone link that can be found on the GitHub repo page, chose a directory for the project, and press clone. This will clone the entire project to a local directory, open the project in Intelli-J, and automatically start building the project with Gradle. All the build steps and project dependencies are fully described in the Gradle build script, and will automatically be installed upon compilation, which Intelli-J will perform before running. To run Kanvas, right click on the Main.kt file and press Run.

To build from the command line, without the Intelli-J IDE, again clone the source code. Then, from the root of the project tree, build the program with Gradle, using the following command: *./gradlew jar*. Finally, run the program on the command line using java, replacing *XXX* with the built version: *java -jar build/libs/kanvas-XXX.jar*.

5.0 Conclusion

Since Kanvas was designed to meet interfaces, not implementations, it was broken down into independent hierarchies and follows robust software design patterns. Kanvas is a clean, easily maintainable, and extensible piece of software. Kanvas notably follows the don't repeat yourself (DRY) principle, the open-close principle, and the single-responsibility principle. These principles, when combined with the architecture of the code, are what allows Kanvas to be so easily expandable. As a challenge to its developers, a foreign programming language, Kotlin, and a contemporary UI framework, JetBrains Compose, were employed to build Kanvas. As such, the development of Kanvas not only exemplifies great design patterns but also illustrates how modern technologies can be applied to further improve software quality.