A black background with a black square

AI-generated content may be incorrect.

CSC 427: Term Project

|  |  |  |
| --- | --- | --- |
|  | **Kareem Hassanein Hassan Abdelkreem** | **320210201** |
| **Eman Saad Mohamed Abbas** | **320210042** |
| **Mariam Fathy Mahmoud Rabia** | **320210262** |
| **Overview** | **Mai Ibrahim AbdElmegeed Ibrahim Kotb** | **320210312** |

The Project is a program to model user interfaces. It allows users to create and simulate user interfaces. The Project provides a user interface, where users can drag and drop UI elements.

## Similar Software

### SceneBuilder

SceneBuilder is a program written in Java that allows designing user interfaces for JavaFX. SceneB- uilder allows users to drag and drop components, edit their properties, style user interfaces using CSS, preview the resulting UI, export user interfaces into FXML and produce skeleton Java code for the controller class.

### wxFormBuilder

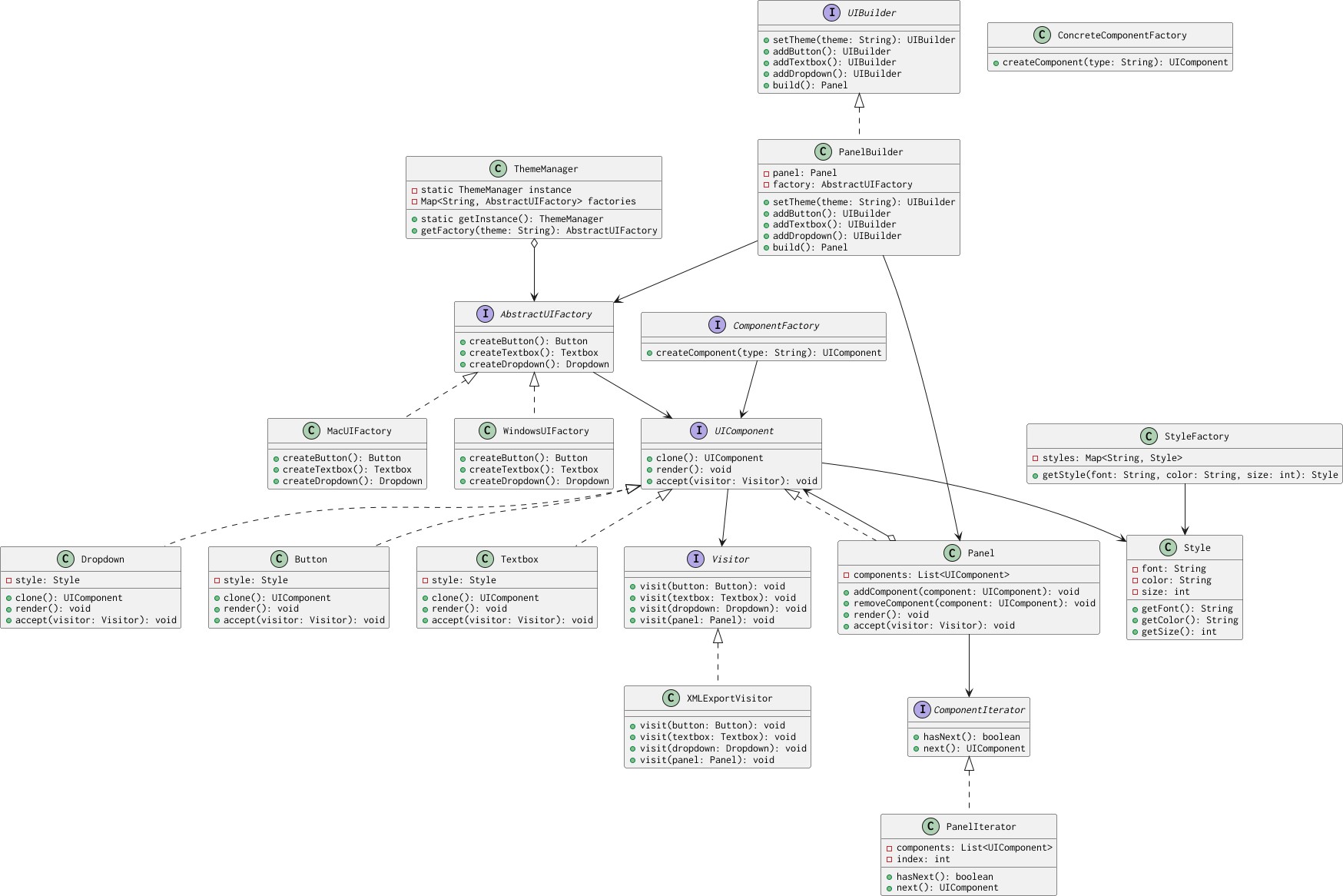
wxFormBuilder is a program written in C++ that allows users to make user interfaces for wxWidgets. It has a similar functionality to SceneBuilder, allowing users to design user interfaces by dragging and dropping components and export the user interface into an XML file. It also allows exporting the UI into multiple languages, like C++, Lua, Python and PHP.

**UML Class Diagram**

The UML class diagram (included in the appendix) illustrates the Modular Design Studio’s architecture and design patterns, as follows:

* **Singleton**: ThemeManager with a static getInstance() and a map of AbstractUIFactory instances.
* **Abstract Factory**: AbstractUIFactory interface, implemented by WindowsUIFactory and MacUIFactory, producing Button, Textbox, and Dropdown.
* **Factory Method**: ComponentFactory interface with ConcreteComponentFactory creating UIComponent instances.
* **Prototype**: UIComponent interface with a clone() method, implemented by Button, Textbox, and Dropdown.
* **Flyweight**: Style class managed by StyleFactory, which maintains a Map<String, Style> for shared styles.
* **Composite**: Panel class aggregating UIComponent instances, with methods like addComponent() and removeComponent().
* **Builder**: UIBuilder interface implemented by PanelBuilder, which uses AbstractUIFactory to construct a Panel.
* **Visitor**: Visitor interface with XMLExportVisitor visiting Button, Textbox, Dropdown, and Panel.
* **Iterator**: ComponentIterator interface implemented by PanelIterator for traversing Panel components.  
  Relationships include composition (Panel to UIComponent), dependency (UIComponent to Style), and realization (e.g., WindowsUIFactory to AbstractUIFactory).

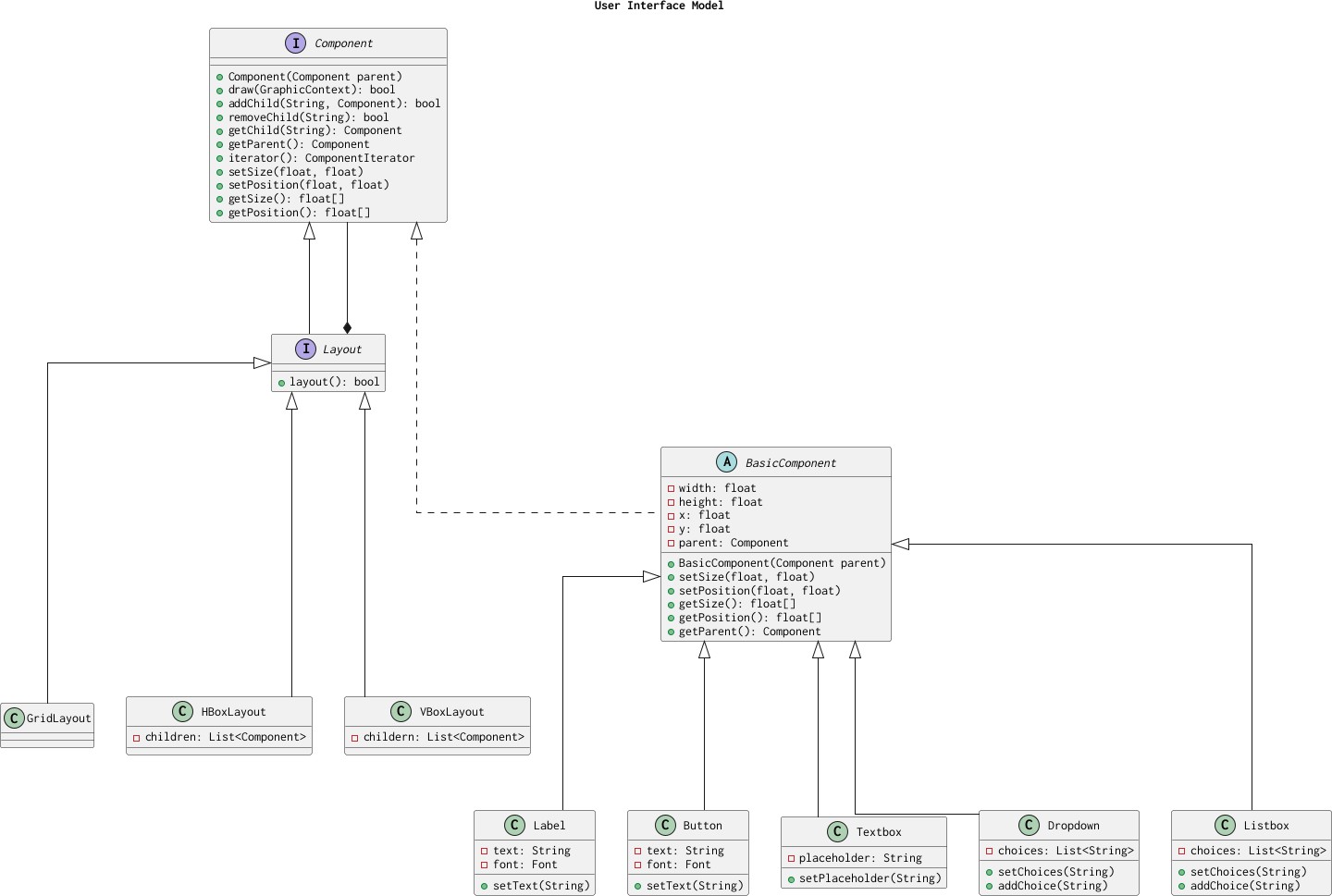
# Patterns Used

****

## User Interface Modeling using Composite Pattern

User interfaces are modeled using a tree. The root of the tree is the entire window. Leaf nodes represent concrete user interface elements like buttons, textboxes and so on, while internal nodes represent containers of other user interfaces elements. Containers often are responsible for determining the size and positioning of their child elements. Containers calculate the positions of their children as offsets from the container’s origin, the size of each children depends on the total size of the container. Each container type have a specific algorithm for defining the locations and sizes of their children. For example, a container type can layout its children in a grid. Another can lay them out in either a row or a column.

User interfaces are constructed by putting UI elements in these containers, and then composing these containers to form a tree. This allows creating complex user interfaces that can handling resizing and layout efficiently. The Composite pattern is the ideal pattern to model this type of composition.



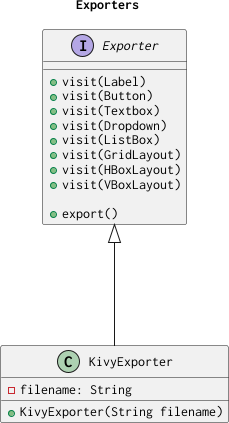
**Singleton Pattern for Theme Management**

The Singleton pattern is implemented in the ThemeManager class, ensuring a single instance manages theme-specific factories. As shown in the UML diagram, ThemeManager has a static instance and a getInstance() method. It maintains a Map<String, AbstractUIFactory> to store factories for themes (e.g., WindowsUIFactory, MacUIFactory). This design centralizes theme management, preventing redundant factory creation and optimizing resource usage.

**Factory Method for Component Creation**

The Factory Method pattern is implemented via the ComponentFactory interface, which defines a createComponent(type: String) method to create UIComponent instances (e.g., Button, Textbox, Dropdown). The ConcreteComponentFactory class, as shown in the UML, implements this method to instantiate components dynamically based on the type parameter. This decouples component creation from client code, enhancing extensibility for adding new component types

**Abstract Factory for Theme-Based Components**

The Abstract Factory pattern is used to create families of theme-specific UI components, as shown in the UML. The AbstractUIFactory interface defines methods (createButton(), createTextbox(), createDropdown()) to produce UIComponent instances. Concrete factories, WindowsUIFactory and MacUIFactory, implement these methods to create components like WindowsButton or MacTextbox. The ThemeManager retrieves the appropriate factory based on the theme, ensuring consistent styling across the UI.

## Abstract Iteration through User Interface Tree Using Iterator Pattern

While modeling user interfaces, it’s sufficient to use a tree-based model. However, there are situations were iteration through all or part of the model is necessary. The iterator pattern allows abstracting the process of looping through the model. Other parts of the software need not know the internal structure of the user interface model in order to iterate through it. It is necessary for the iterator to return components in a well-defined sequence (often a pre-order traversal of the tree).

## 

## Define Exporters using Visitor Pattern

The Visitor pattern is used to encapsulate the logic of exporting the user interface into a file. The visitor pattern isolates the logic that operates on the UI model from the representation of the model. This isolation allows extending the code that operates on the UI model, while keeping the UI model code simple and easy to maintain. For each file type that’s targeted, there’s a concrete visitor class that implements exporting to that file type.

## Share Resources using Flyweight Pattern

Designing user interfaces needs resources like fonts, icons and images. The flyweight pattern is used to represent these resources. This allows efficient use of memory. Assets are represented as an object per asset, and UI elements refer to that instance of the asset in order to render itself.

A screen shot of a computer code

AI-generated content may be incorrect.

**Cloning UI Components using Prototype Pattern**

A screenshot of a computer program

AI-generated content may be incorrect.The Prototype pattern is used to create copies of UI components, such as buttons, textboxes, and dropdowns, by cloning existing instances. Each UI component implements a `clone()` method, which returns a new instance with the same properties and style as the original. This approach is particularly useful when users duplicate elements in the design interface, as it avoids the overhead of creating new components from scratch. By leveraging the Prototype pattern, FigmaKiller ensures efficient component replication while maintaining consistency in styling and behavior across duplicated elements.

## Building a User Interface using Builder Pattern

User interfaces are often a large and complex tree of components and containers. Therefore, the builder pattern is used to encapsulate the logic of building a UI tree. The builder is used to read a UI design from a save file. Using the builder pattern allows the logic of building a UI and the representation of the UI to vary independently.

# Interactions among Patterns

## Exporting using Iterator, Factory Method and Visitor

In order to export the user interface to a file, the program needs to iterate through the tree structure of the user interface. The iterator method abstracts the iteration through the structure. The visitor pattern access the iterator through a factory method, by calling iterator() on the root element. The visitor then loops through all the components using the iterator, accumulating state. The client code then calls a export() method to export to a file.

## Building a User Interface Model using Builder and Composite

The builder pattern is used to separate the logic of building a user interface, from the logic that represents it. The builder builds a user interface tree by methods that adds components like buttons and textboxes and containers for those components. The builder is used to gradually build the user interface, allowing for the creation of complex interfaces with maintainable code.

## Managing Resources using Singleton and Flyweight

The program uses resources like fonts and icons. These resources need to be managed carefully to avoid exhausting memory. The flyweight pattern is used to share instances of resources like fonts and icons. These resources are only loaded once, when needed, and components that use these resources refer to the same shared instance of that resource. Loading resources is done by the Singleton class ResourceManager. ResourceManager maintains a map of loaded resources. Each resource is identified by its filename. When a component requests a resource, ResourceManager consults its maps. If the resource is found, a reference to it is returned. Otherwise, the resource is loaded, stored in the map, and then returned.