National University "Kyiv-Mohyla Academy"

LABORATORY WORK #2

The work was performed by:

- Hnutov Oleksandr Volodymyrovych
- Verstiuk Myroslav Oleksandrovych

AWP64

AWP64 (Automated workplace) is a utility for automating work of a small enterprise in working with a warehouse, created using Swelm.

Formulation of the problem

It is necessary to automate the work of a small enterprise in working with a warehouse.

There are several groups of goods (for example: Food, non-food...). In each group of goods there are specific goods (for example: flour, buckwheat ...). Each product has the following properties - name, description, manufacturer, quantity in stock, price per unit. A group of goods contains the following properties - name, description.

Implement

- Implement a graphical user interface
- Save data to a file/files. One of the options: There is a file that contains the names of all product groups. Products from each product group are in a separate file.
- The name of the product is unique (it cannot be found in any group of products).
- The name of the product group is unique.
- Add/edit/delete a group of products - when deleting a group of products, delete all products.
- Implement the addition/editing/deletion of a product in a group of products (meaning name, description, manufacturer, price per unit).
- Implement the product addition interface (buckwheat groats arrived in the warehouse - 10 pieces), the product write-off interface (5 buckwheat groats were sold)
- Product search.
- Output of statistical data: output of all goods with information by warehouse, output of all goods by group of goods with information, total cost of goods in stock (quantity * per price), total cost of goods in a group of goods.
- Add a work performance report with a description of the distribution of roles to the work.

Distribution of roles in the group

Oleksandr Hnutov	Myroslav Verstiuk
Swelm UI framework components and concepts	Swelm UI widgets
Backend layer and data models	AWP64 widgets
Documentation	Frontend presentation layer
Refactoring and edits	UI/UX Design
Splashscreen design	

Implemented opportunities

1. **Product Management**:
    - View and edit product details.
    - Add new products.
    - Delete existing products.
    - Open products for detailed editing.
    - Display products with customizable views.
2. **Group Management**:
    - View and edit group details.
    - Add new groups.
    - Delete existing groups.
    - Open groups for detailed editing.
    - Display groups with customizable views.
3. **Manufacturer Management**:
    - View and manage manufacturers.
4. **Search Functionality**:
    - Perform searches for products, groups, and other entities.
    - Utilize search patterns for advanced search operations.
5. **Screen Navigation**:
    - Navigate between different screens within the application.
    - Main screen for overview and access to other functionalities.
    - Edit screens for modifying product and group details.
6. **Custom UI Widgets**:
    - Display various components such as products, groups, search bars, cards, icons, etc.
    - Labeled text input fields for clear data entry.
    - Main menu for easy access to different parts of the application.
7. **Error Handling**:
    - Handle errors and exceptions related to product and group management.
    - Notify users of invalid inputs or processing failures.
8. **Messaging System**:

- Communicate messages between different parts of the application.
- Utilize message classes for specific actions like saving, deleting, or editing entities.

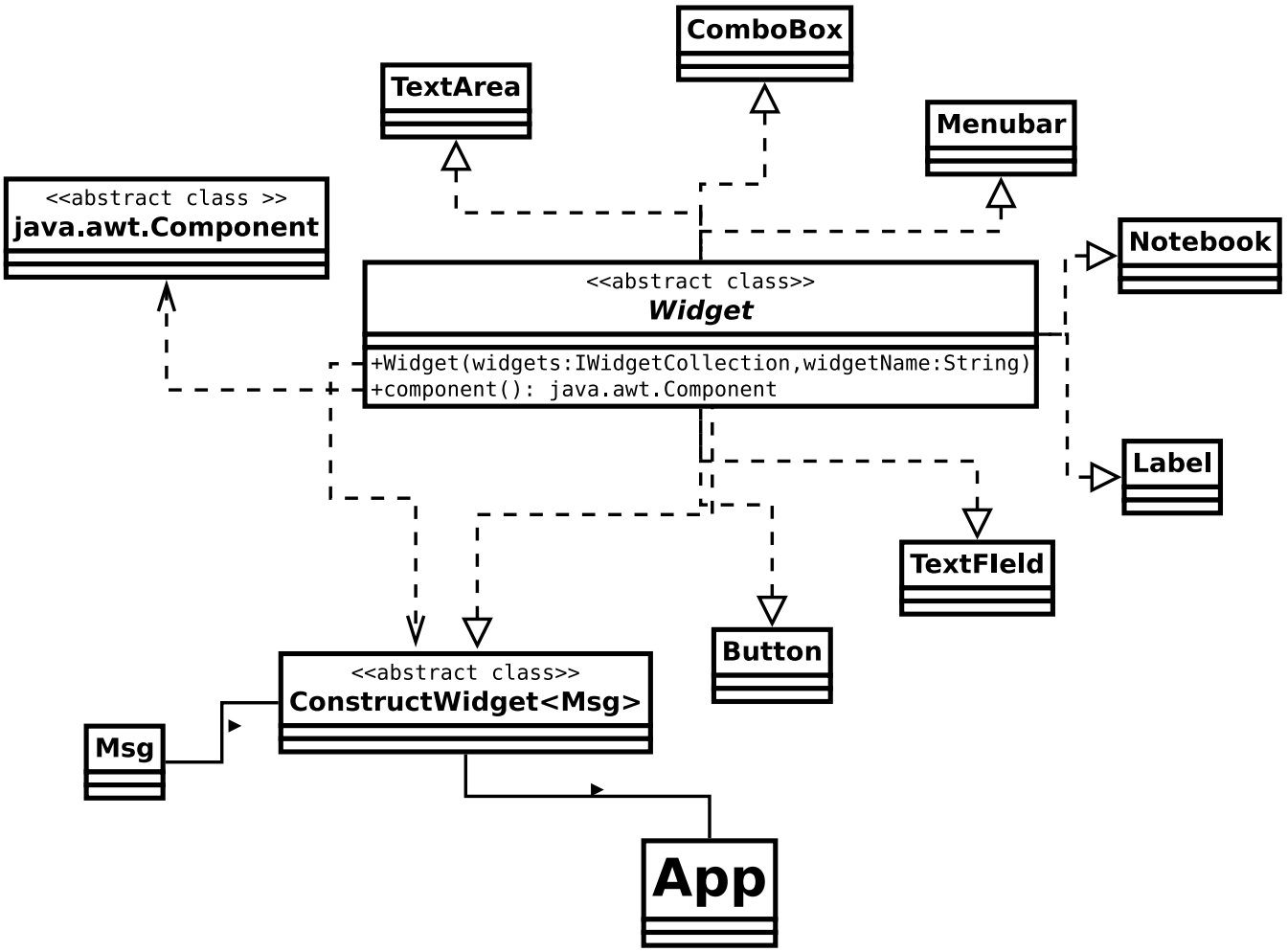
9. **Model Management**:
   - Centralize data management and business logic within the AwpModel class.
   - Manage entities such as products, groups, and manufacturers.
10. **Modular Architecture**:
    - Organize the application into separate packages for views, screens, models, and messages.
    - Follow a modular structure for easier maintenance and scalability.
11. **Customizable Views**:
    - Customize the display of products and groups according to specific requirements.
    - Utilize different UI components and layouts for flexible and appealing interfaces.

The structure of the program. UML

This chapter contains a description of a structure of AWP and Swelm UI libraries

Swelm UI Framework Overview:

The Swelm UI framework is a Java-based framework built on top of Swing. It aims to provide developers with a comprehensive set of tools and components for creating declarative user interfaces for Automated Workplace (AWP) projects. The framework follows a modular structure, with distinct packages for different types of components and utilities.



Package Structure

- swelm.app:
    - **App**: Main entry point for Swelm applications.
    - **Window**: Class for managing application windows.
    - **Splashcreen**: Implementation of a splash screen for application startup.
    - **IWidgetCollection**: Interface for managing collections of widgets.
    - **Widgets**: Utility class for working with widgets.
- swelm.widgets:

    - **Widget**: Base class for all UI widgets.
    - **Tab**: Implementation of a tabbed pane.
    - **ScrollablePanel**: Panel with scrollbars for displaying large content.
    - **TextField**: Text input field.
    - **HTMLLabel**: Label component for displaying HTML content.
    - **Button**: Implementation of a clickable button.
    - **Image**: Component for displaying images.
    - **Notebook**: Container for organizing multiple panels in a notebook-like interface.
    - **TextArea**: Multiline text input field.
    - **ConstructWidget**: Interface for constructing widgets dynamically.
    - **TextPane**: Component for displaying styled text.

- swelm.widgets.menu:
    - **Menu**: Implementation of a menu component.
    - **MenuItem**: Menu item component.
    - **SingleItem**: Interface for single menu items.
    - **MenuBar**: Menu bar component for organizing menus.

- swelm.widgets.containers:
    - **CoreWidget**: Base class for container widgets.
    - **BorderContainer**: Container with border layout.
    - **Glue**: Utility class for adding flexible space.
    - **GridContainer**: Container with grid layout.
    - **BoxContainer**: Container with box layout.
    - **WrapContainer**: Container with wrap layout.

- swelm.widgets.swing:
    - **RoundedBorder**: Implementation of a rounded border for Swing components.
    - **HTMLJLabel**: JLabel component for displaying HTML content.
    - **JPlaceholderTextField**: Text field component with placeholder text support.
    - **WrapLayout**: Layout manager for wrapping components.
    - **JPlaceholderTextArea**: Text area component with placeholder text support.
    - **TruncatingTextPane**: Text pane component with truncation support.

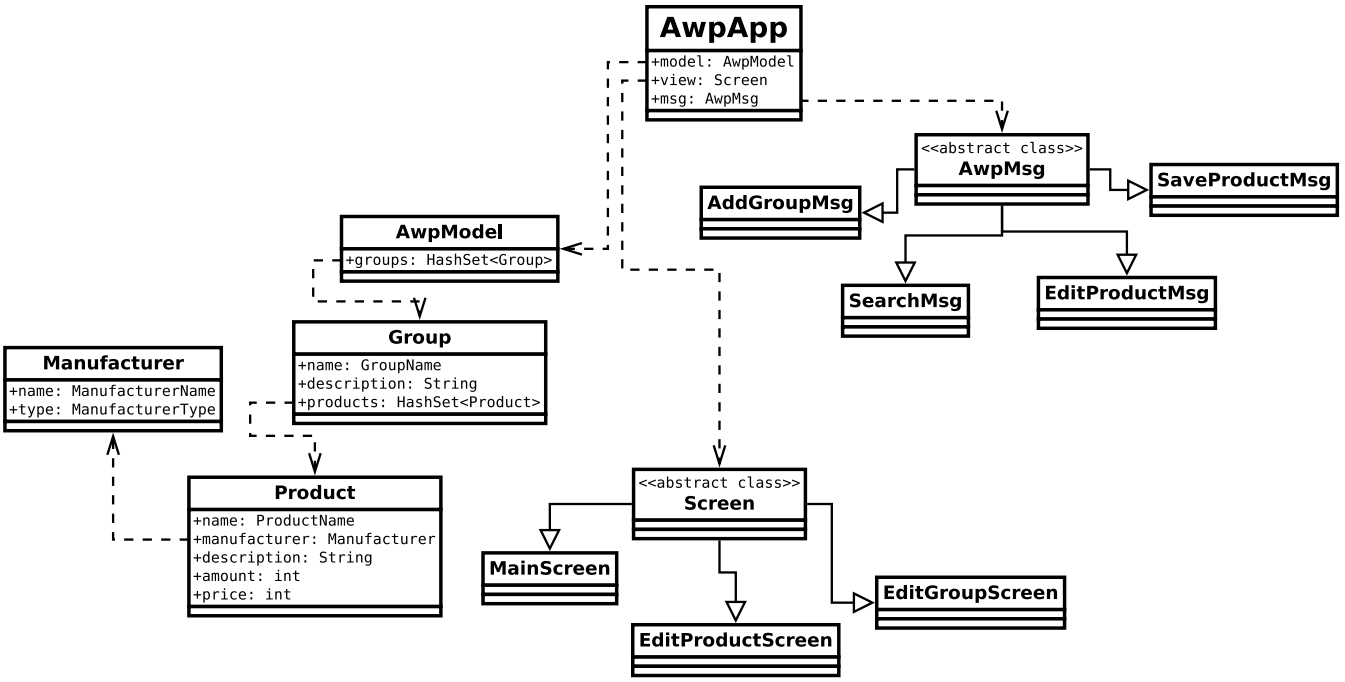
- swelm.widgets.ui:
    - **SwelmTextFieldUI**: UI delegate for customizing text field appearance.
    - **SwelmTextAreaUI**: UI delegate for customizing text area appearance.
    - **SwelmTabbedPaneUI**: UI delegate for customizing tabbed pane appearance.

- swelm.utils:

- **ResourceImage**: Utility class for working with image resources.
- **Pos.java, Color.java, Size**: Utility classes for positioning, color, and size manipulation.
- **DumpMode.java, UnitMsg.java, MsgBox**: Utility classes for debugging and messaging.
- **AdjustableWidget**: Interface for adjustable widgets.

AWP Project Overview

The Automated Workplace (**AWP**) project is a Java application designed to streamline and automate various tasks related to workplace management. It encompasses functionalities for managing products, groups, manufacturers, and search operations. The project follows a modular architecture, with distinct packages for different layers of the application, including views, screens, models, and messages.



Package Structure:

- **awp.view**:
    - **AwpApp**: Main entry point for the AWP application.
    - **widgets**: Package containing custom UI widgets used throughout the application.
        - **DisplayProducts**: Widget for displaying products.
        - **SearchBar**: Widget for search functionality.
        - **Card**: Custom card component.
        - **Icon**: Icon component.
        - **DisplayGroup**: Widget for displaying groups.
        - **MainMenu**: Main menu component.
        - **LabeledTextArea, LabeledTextField**: Components for labeled text input.
        - **DisplayItem**: Widget for displaying individual items.
        - **AddGroup**: Widget for adding groups.
- **awp.view.screens**:

    - **Screen**: Base class for all screens in the application.
    - **EditProductScreen**: Screen for editing product details.
    - **MainScreen**: Main screen of the application.
    - **EditGroupScreen**: Screen for editing group details.

- **awp.model**:

    - **AwpModel**: Main model class managing data and business logic.
    - **Group, Manufacturer, Product**: Model classes representing entities in the application domain.
    - **GroupName, ManufacturerName, ProductName**: Wrapper classes for names of entities.
    - **search**: Package containing classes related to search functionality.
        - **SearchPattern**: Class representing search patterns.

- **awp.model.error**: Classes representing various error conditions and exceptions that can occur during model processing. **InvalidProductNameException, InvalidGroupNameException, InvalidSearchPromptException, ModelProcessingException, InvalidManufacturerNameException, NoSuchGroupException**.

- **awp.msg**: Classes representing different types of messages used for communication within the application. **SaveGroupMsg, DeleteProductMsg, NewGroupMsg, OpenProductMsg, AwpMsg, AddProductMsg, SearchMsg, DeleteGroupMsg, AddGroupMsg, EmptyMsg, SaveProductMsg, EditGroupMsg, EditProductMsg, OpenGroupMsg**.

Data structure

```
{
  "groups" : [ {
    "name" : {
      "name" : "group name"
    },
    "description" : "group description",
    "products" : [ {
      "name" : {
        "name" : "product name"
      },
      "manufacturer" : {
        "type" : "manufacturer type",
        "name" : {
          "name" : "manufacturer name"
        }
      },
      "description" : "product description",
      "amount" : 123,
      "price" : 456
    }
  }
}
```

Solving the main tasks

Tasks Solvation:

1. **Implement a Graphical User Interface (GUI)**:
    - Develop a user-friendly GUI using own Swelm UI framework, providing intuitive navigation and interaction for users.
2. **Save Data to File/Files**:
    - Implement data serialization to save product groups and products to JSON file.
3. **Unique Product and Group Names**:
    - Ensure uniqueness of product names and product group names to prevent duplication and maintain data integrity.
    - Use "Newtype" paradigm to ensure safety and types validation
4. **Group Management (Add/Edit/Delete)**:
    - Implement functionality to add, edit, and delete product groups.
    - Upon deletion of a product group, remove all associated products.
    - Use HashSets to store data
5. **Product Management (Add/Edit/Delete)**:
    - Enable users to add, edit, and delete products within product groups, providing options to modify attributes such as name, description, manufacturer, and price per unit.
    - Use HashSets to store data
6. **Product Search**:
    - Develop a search feature enabling users to search for products based on various criteria such as name, description, or manufacturer.
    - Use regular expressions to perform fast and errorless search
7. **Statistical Data Output**:
    - Provide functionality to output statistical data, including:
        - List of all products with warehouse information.
        - List of all products by product group with details.
        - Calculation of total cost of goods in stock (quantity * unit price).
        - Calculation of total cost of goods in a specific product group.

Problems at work

- **Problem**: Data Management Complexity
- **Solution**: Using *FasterXML* library to store nested groups/products data as a JSON file

---

- **Problem**: Products names uniqueness and validation
- **Solution**: Using Rust-derived pattern "*newtype*", which gives you the opportunity to validate data during object creation, but not using it; using `HashSet`s with lambda expressions to fastly search for a product/group to ensure its uniequeness

---

- **Problem**: *Swing UI* development complexity and bad user experience
- **Solution**: Developing own declarative UI framework Swelm based on Swing and inspired by *Flutter* and *Relm* frameworks

---

- **Problem**: Search Functionality Optimization
- **Solution**: Using regular expressions to ensure correct search queries and using lambda expressions and HashSets for a fast search

---

- **Problem**: Using multiple working context in a single app
- **Solution**: Implementing `Screen` abstract class to easily switch working context without data loss

---

Conclusion

In conclusion, our journey through the development of the workplace management application has been marked by a series of challenges and solutions aimed at creating an efficient and user-friendly tool. We identified complexities in data management, such as organizing files for product groups and individual products, and addressed them with serialization techniques and robust validation mechanisms to ensure data integrity. The necessity of maintaining unique product and product group names was met with comprehensive validation strategies to prevent duplication and maintain consistency throughout the system.

Moreover, the implementation of transaction tracking mechanisms and the optimization of search functionality were pivotal in enhancing the application's usability and effectiveness. By providing intuitive interfaces and efficient search capabilities, we empowered users to navigate the system effortlessly and access information swiftly. Additionally, our focus on generating comprehensive statistical reports enables users to gain valuable insights into inventory management and business performance, driving informed decision-making.

In essence, our project represents a culmination of efforts to overcome various challenges in workplace management through innovative solutions and diligent teamwork. As we conclude this phase, we recognize the importance of continual improvement and adaptation to meet evolving needs. With a solid foundation in place and a commitment to excellence, our workplace management application stands ready to streamline operations and empower users in their daily tasks.

Appendix

Source code is located in GitHub repository and is licensed under permissive **Unlicense license**

Swelm UI

Swelm is a Swing-based Java UI framework inspired by flutter and relm projects.

Guide

This chapter provides a basic guide to create a simple application using swelm framework

Creating app

In swelm every application is a widget, which is constructed from other atomic or complex widgets. To create an application create a class, which extends `ConstructWidget`:

```
public class MyApp extends ConstructWidget<MyMsg> {

    @Override
    public void init() {
        // Initialization
    }

    @Override
    public void event(MyMsg msg) {
        // Event processing
    }

    @Override
    public Widget build() {
        // Building out application
    }

}
```

Now let's figure out every component of our application:

- `MyMsg` is our custom class of message, which is a component generated by our application in response to the user's interaction. It is process in `event` method. Messages can be of any type, but we'll talk about it more precisely in chapter Events
- `init` method is executed once after building the application to setup and manipulate application widgets at the beginning. Read more about it in Accessing widgets chapter.
- `build` method is where our application view arises. There we place our widgets in a structure we want it to be shown. Such an approach is called Declarative programming

Let's add `CoreWidget` to our build method. In a constructor we pass our application instance (we can also pass other collection which implements `IWidgetCollection` interface, but it is not recomended; read more about it in Accessing widgets chapter), unique ID (which is usually its underscored name) and an inner widget, whichin most cases is a container, where other widgets will be stored:

```
@Override
public Widget build() {
    return new CoreWidget(this, "my_app",
        // inner widget
    );
}
```

Add a `BoxContainer` widgets which will represent our application main layout:

```
@Override
public Widget build() {
    return new CoreWidget(this, "my_app",
        new BoxContainer(this, "my_box")
    );
}
```

Now it is empty. Let's add some widgets to it, using its `children` parameter. All widgets in Swelm are constructed autonomously and their parameters are accessed with methods:

```
@Override
public Widget build() {
    return new CoreWidget(this, "my_app",

        new BoxContainer(this, "my_box")
            .children(new Widget[]{
                // Our main widgets
            })

    );
}
```

Now add two buttons with labels "Button1" and "Button2" to our box. We are doing it the same way, using `text` parameter:

```
@Override
public Widget build() {
    return new CoreWidget(this, "my_app",

        new BoxContainer(this, "my_box")
            .children(new Widget[]{
                new Button(this, "button1")
                    .text("Button1"),

                new Button(this, "button2")
                    .text("Button2")
            })

    );
}
```

Also let's say we want our box to be aligned vertically. The same way we use `align` parameter to our `BoxContainer`:

```
@Override
public Widget build() {
    return new CoreWidget(this, "my_app",

        new BoxContainer(this, "my_box")
            .align(BoxAlign.Vertical)
            .children(new Widget[]{

                new Button(this, "button1")
                    .text("Button1"),

                new Button(this, "button2")
                    .text("Button2")

            })

    );
}
```

Congratulations! You've just created your first swelm application. But how to run it? For this purpose we'll use `App` class:

```
public static void main(String[] args) {
    new App<MyMsg>(new MyApp())
        .run();
}
```

So for now our application looks like this:


My app
Button1
Button2

Accessing widgets

We've created an application and now we want to access our widgets somehow. For example, we would want to check some config parameter at the application initialization to find out, whether we must enable or disable some widget.

Namely for this purpose we have passed our application (or rarely other IWidgetCollection) instance when creating our widgets:

```
new Button(this, "my_button")
    .text("MyButton"),
```

So now we can get it from our application in `init` or `event` methods by its unique ID:

```
@Override
public void init() {
    Button myButton = (Button) getWidget("my_button");
    if (config.someParam == 0)
        myButton.setVisible(false);
    else
        myButton.setVisible(true);
}
```

Or alternatively we can create another `IWidgetCollection` instance inside our application and use it directly, but this approach is not recommended:

```
public class MyApp extends ConstructWidget<UnitMsg> {

    private Widgets myWidgets = new Widgets();

    @Override
    public void init() {
        ((Button) myWidgets.getWidget("my_button")).setVisible(true);
    }

    @Override
    public Widget build() {
        return new CoreWidget(this, "my_app",
            new BorderContainer(this, "my_border")
                .north(new Button(myWidgets, "my_button")
                    .text("MyButton"))
        );
    }

    public static void main(String[] args) {
        new App<UnitMsg>(new MyApp())
            .run();
    }

}
```

NOTE: Every `ConstructWidget` can also be `IWidgetCollection`

Events

Events in Swelm are processed using **messages**. They are objects, which come from user interactions events and represent the current state of the application; For easy messages processing we recommend using `enums` :

```
enum MyMsg {
    DoSomethingCool,
    DoSomethingCool2,
    ExitApplication,
}
```

Usage example

```
enum MyMsg {
    HelloWorld,
    NotHelloWorld,
}

public class MyApp extends ConstructWidget<MyMsg> {

    @Override
    public void event(MyMsg msg) {
        switch (msg) {
            case HelloWorld:
                System.out.println("Hello world!");
                break;

            default:
                break;
        }
    }

    @Override
    public Widget build() {
        return new CoreWidget(this, "my_app",
            new BorderContainer(this, "my_border")
                .north(new Button(this, "my_button")
                    .text("Say hello")
                    .clicked(this, MyMsg.HelloWorld)
                )
        );
    }

    public static void main(String[] args) {
        new App<MyMsg>(new MyApp())
            .run();
    }

}
```

Creating own widget

Creating a widget in Swelm is similar to creating an application (because for both we use `ConstructWidget` class). As for application we must specify its message type. If our widget won't produce any events, we can use `UnitMsg` type. Else we can specify any other message type or application's one:

```
public class MyWidget extends ConstructWidget<MyMsg> {

    // We must save our app instance in order to
    // spawn widgets to it
    private MyApp app;

    public MyWidget(MyApp app, String widgetName) {
        super(app, widgetName);
        this.app = app;
    }

    @Override
    public void event(MyMsg msg) {
        // Process event
    }

    @Override
    public Widget build() {
        return new BoxContainer(app, "my_box")
            .children(new Widget[]{
                new Button(app, "my_button")
                    .text("Button1")
                    .clicked(this, MyMsg.HelloWorld)
            });
    }
}
```

As you can see, we can't use more than 1 such widget in our application, because its widgets IDs will be the same for every instance of the widget. In order to avoid this, we can use our `widgetName` as a prefix for our widgets:

```
public class MyWidget extends ConstructWidget<MyMsg> {

    private MyApp app;
    private String widgetName;

    public MyWidget(MyApp app, String widgetName) {
        super(app, widgetName);
        this.app = app;
        this.widgetName = widgetName;
    }

    @Override
    public void event(MyMsg msg) {
        // Process event
    }

    @Override
    public Widget build() {
        return new BoxContainer(app, widgetName+"_my_box")
            .children(new Widget[]{
                new Button(app, widgetName+"_my_button")
                    .text("Button1")
                    .clicked(this, MyMsg.HelloWorld)
            });
    }
}
```

And now we can process our message in the widget or simply pass it directly to the application:

```
public class MyWidget extends ConstructWidget<MyMsg> {

    private MyApp app;
    private String widgetName;

    public MyWidget(MyApp app, String widgetName) {
        super(app, widgetName);
        this.app = app;
        this.widgetName = widgetName;
    }

    @Override
    public void event(MyMsg msg) {
        app.event(msg);
    }

    @Override
    public Widget build() {
        return new BoxContainer(app, widgetName+"_my_box")
            .children(new Widget[]{
                new Button(app, widgetName+"_my_button")
                    .text("Button1")
                    .clicked(this, MyMsg.HelloWorld)
            });
    }
}
```

That's all! Now we can use our widget in the application:

```java
enum MyMsg {
    HelloWorld,
    DoSomething,
}

public class MyApp extends ConstructWidget<MyMsg> {

    @Override
    public void event(MyMsg msg) {
        switch (msg) {
            case HelloWorld:
                System.out.println("Hello world!");
                break;

            default:
                break;
        }
    }

    @Override
    public Widget build() {
        return new CoreWidget(this, "my_app",
            new BorderContainer(this, "my_border")
                .north(new MyWidget(this, "my_widget"))
        );
    }

    public static void main(String[] args) {
        new App<MyMsg>(new MyApp())
            .run();
    }

}
```

Example: Counter

Now let's create a real working example: it will be an application with two buttons - increment and decrement, and a label with a number, which will be manipulated.

Let's start with a carcass - application base and message:

```
enum CounterMsg {
    Increment,
    Decrement,
}

public class CounterApp extends ConstructWidget<CounterMsg> {

    // Our value to modify
    int i = 0;

    @Override
    public void event(CounterMsg msg) {
        // Event processing
    }

    @Override
    public Widget build() {
        return new CoreWidget(this, "counter_app",
            // Our widgets
        );

    }

    // Initialize application
    public static void main(String[] args) {
        new App<CounterMsg>(new CounterApp())
            .size(800, 100)
            .run();
    }

}
```

Now create a border container, where we will put our buttons and a label:

```
@Override
public Widget build() {
    return new CoreWidget(this, "counter_app",

        new BorderContainer(this, "border")
            .west(

                // Increment button, which sends `CounterMsg.Increment`
message
                new Button(this, "inc_button")
                    .text("Increment")
                    .clicked(this, CounterMsg.Increment)

            )
            .center(

                // We will use box container with glues to center our label
                new BoxContainer(this, "label_box").children(new Widget[]{

                    new Glue(this, "glue1", Orientation.Horizontal),

                    new Label(this, "label")
                        .text(String.valueOf(i)),

                    new Glue(this, "glue2", Orientation.Horizontal),

                })

            )
            .east(

                // Decrement button, which sends `CounterMsg.Decrement`
message
                new Button(this, "dec_button")
                    .text("Decrement")
                    .clicked(this, CounterMsg.Decrement)

            )
    );

}
```

And now let's process our sent messages using `switch` statements:

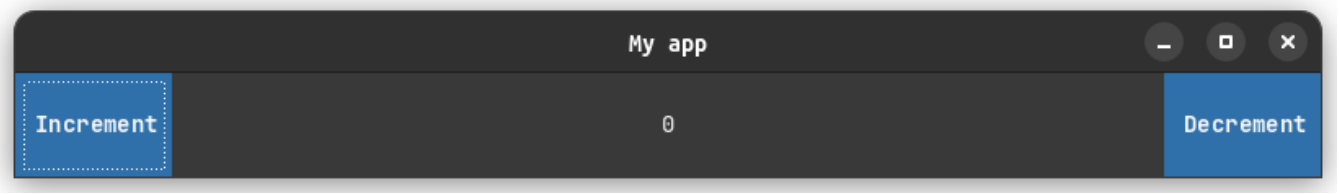
```
@Override
public void event(CounterMsg msg) {
    switch (msg) {
        case Increment:
            // Increment our value and set label text to it
            i++;
            ((Label) getWidget("label")).setText(String.valueOf(i));
            break;

        case Decrement:
            // Decrement our value and set label text to it
            i--;
            ((Label) getWidget("label")).setText(String.valueOf(i));
            break;

        default:
            break;
    }
}
```

So our application code will be the following:

```
import org.verstiukhnutov.swelm.app.App;
import org.verstiukhnutov.swelm.widgets.Button;
import org.verstiukhnutov.swelm.widgets.ConstructWidget;
import org.verstiukhnutov.swelm.widgets.CoreWidget;
import org.verstiukhnutov.swelm.widgets.Label;
import org.verstiukhnutov.swelm.widgets.Widget;
import org.verstiukhnutov.swelm.widgets.containers.BorderContainer;
import org.verstiukhnutov.swelm.widgets.containers.BoxContainer;
import org.verstiukhnutov.swelm.widgets.containers.Glue;
import org.verstiukhnutov.swelm.widgets.containers.Glue.Orientation;

enum CounterMsg {
    Increment,
    Decrement,
}

public class CounterApp extends ConstructWidget<CounterMsg> {

    int i = 0;

    @Override
    public void event(CounterMsg msg) {
        switch (msg) {
            case Increment:
                i++;
                ((Label) getWidget("label")).setText(String.valueOf(i));
                break;

            case Decrement:
                i--;
                ((Label) getWidget("label")).setText(String.valueOf(i));
                break;

            default:
                break;
        }
    }

    @Override
    public Widget build() {
        return new CoreWidget(this, "counter_app",

            new BorderContainer(this, "border")
                .west(

                    new Button(this, "inc_button")
                        .text("Increment")
                        .clicked(this, CounterMsg.Increment)

                )
                .center(

                    new BoxContainer(this, "label_box").children(new Widget[]
{

                        new Glue(this, "glue1", Orientation.Horizontal),
```

```
                        new Label(this, "label")
                            .text(String.valueOf(i)),

                        new Glue(this, "glue2", Orientation.Horizontal),

                    })

                )
                .east(

                    new Button(this, "dec_button")
                        .text("Decrement")
                        .clicked(this, CounterMsg.Decrement)

                )
        );

    }

    public static void main(String[] args) {
        new App<CounterMsg>(new CounterApp())
            .size(800, 100)
            .run();
    }

}
```

The final application will look like this



Widgets

Swelm provides a range of widgets for building user interfaces. These widgets include buttons for triggering actions, text fields for user input, and containers for organizing layout, such as panels and containers. Each widget offers specific functionality to enhance the user experience and customize the application interface.

Each widget implements `AdjustableWidget` interface, so you can adjust every Swelm widget's size, color etc.

CoreWidget

The CoreWidget class in Swelm is a fundamental building block for constructing widgets within your application. It serves as a container for other widgets, allowing you to organize and manage the layout of your application's user interface.

```
public CoreWidget(IWidgetCollection widgets, String widgetName, Widget child)
```

- widgets: An instance of IWidgetCollection interface, which provides access to the collection of widgets within the application.
- widgetName: A unique identifier for the widget.
- child: The inner widget to be contained within the CoreWidget.

Usage

To utilize the CoreWidget class, you can create an instance of it within your application's code and specify the inner widget to be contained. Here's an example of how to create a CoreWidget:

```
@Override
public Widget build() {
    return new CoreWidget(this, "my_core_widget",

        new Button(this, "button1")
            .text("I am the only widget inside CoreWidget!"),

    );
}
```

In this example, we create a new CoreWidget named "my_core_widget" and add a Button widget as its inner child;

Button

The `Button` class in Swelm is a widget that represents a clickable button in your application's user interface.

```
public Button(IWidgetCollection widgets, String widgetName)
```

- widgets: An instance of IWidgetCollection interface, which provides access to the collection of widgets within the application.
- widgetName: A unique identifier for the widget.

Main methods

`text(String text)`

Sets the text label for the button.

```
public Button text(String text)
```

- text: The text to be displayed on the button.

`clicked(ConstructWidget<Msg> app, Msg msg)`

Registers an action listener to handle button clicks.

```
public <Msg> Button clicked(ConstructWidget<Msg> app, Msg msg)
```

- app: An instance of ConstructWidget representing the application.
- msg: The message to be processed when the button is clicked.

`setText(String text)`

Sets the text label for the button.

```
public void setText(String text)
```

- text: The text to be displayed on the button.

Usage

To create a button in your Swelm application, you can instantiate a `Button` object and customize its properties using the available methods. Here's an example:

```
Button myButton = new Button(this, "my_button")
    .text("Click me")
    .clicked(this, new MyMsg("Button clicked"));
```

In this example, we create a button with the text "Click me", positioned at (100, 100) with a size of 100x50 pixels. We also register an action listener to handle button clicks and send a message `MyMsg("Button clicked")` to the application when clicked.

Notebook

The `Notebook` class in Swelm is a widget that represents a notebook or tabbed pane in your application's user interface. It allows you to organize content into multiple tabs, with each tab containing a distinct set of components.

```
public Notebook(IWidgetCollection widgets, String widgetName)
```

- widgets: An instance of IWidgetCollection interface, which provides access to the collection of widgets within the application.
- widgetName: A unique identifier for the widget.

Main methods

```
tabs(Tab[] tabs)
```

Adds tabs to the notebook.

```
public Notebook tabs(Tab[] tabs)
```

- tabs: An array of Tab objects representing the tabs to be added to the notebook.

Usage

To create a notebook in your Swelm application, you can instantiate a `Notebook` object and add tabs to it using the `tabs()` method. Here's an example:

```
Notebook myNotebook = new Notebook(this, "my_notebook")
    .bounds(100, 100, 300, 200)
    .tabs(new Tab[]{
        new Tab("Tab 1", new Panel(this, "panel1", new Button(this,
"my_button").text("Click me"))),
        new Tab("Tab 2", new Panel(this, "panel2", new Button(this,
"my_button").text("Click me")))
    });
```

In this example, we create a notebook positioned at (100, 100) with a size of 300x200 pixels. We add two tabs to the notebook, each with a label ("Tab 1" and "Tab 2") and a corresponding panel.

Notes

- Notebook provides a convenient way to organize and present multiple sets of content within a single container.
- It utilizes a JTabbedPane from the Swing library to implement tabbed functionality.

Tab

The `Tab` class in Swelm represents a single tab within a notebook or tabbed pane. Each tab consists of a label and an associated panel containing the tab's content.

```
public Tab(String label, Panel panel)
```

- label: The label or title of the tab.
- panel: An instance of the Panel class representing the content of the tab.

Methods

`getLabel()`

Returns the label of the tab.

```
public String getLabel()
```

- Returns: The label of the tab.

`getPanel()`

Returns the panel associated with the tab.

```
public Panel getPanel()
```

- Returns: An instance of the Panel class representing the content of the tab.

Panel

The `Panel` class in Swelm is a widget that represents a graphical container for other components within your application's user interface. It provides a way to group and organize components together.

```
public Panel(IWidgetCollection widgets, String widgetName, Widget child)
```

- widgets: An instance of IWidgetCollection interface, which provides access to the collection of widgets within the application.
- widgetName: A unique identifier for the widget.
- child: The inner widget to be contained within the Panel.

Usage

To create a panel in your Swelm application, you can instantiate a `Panel` object and add child components to it. Here's an example:

```
Panel myPanel = new Panel(this, "my_panel", new Button(this,
"my_button").text("Click me"));
```

In this example, we create a panel named "my_panel" and add a button as its child component.

Notes

- Panel provides a way to group and organize components within a container.
- It utilizes a JPanel from the Swing library to implement panel functionality.

TextField

The `TextField` class in Swelm is a widget that represents a text input field in your application's user interface. It allows users to enter and edit text.

```
public TextField(IWidgetCollection widgets, String widgetName)
```

- widgets: An instance of IWidgetCollection interface, which provides access to the collection of widgets within the application.
- widgetName: A unique identifier for the widget.

Main methods

`text(String text)`

Sets the text content of the text field.

```
public TextField text(String text)
```

- text: The text to be displayed in the text field.

`placeholder(String placeholder)`

Sets the placeholder text for the text field.

```
public TextField placeholder(String placeholder)
```

- placeholder: The placeholder text to be displayed when the text field is empty.

`setText(String text)`

Sets the text content of the text field.

```
public void setText(String text)
```

- text: The text to be displayed in the text field.

`setPlaceholder(String placeholder)`

Sets the placeholder text for the text field.

```
public void setPlaceholder(String placeholder)
```

- placeholder: The placeholder text to be displayed when the text field is empty.

Usage

To create a text field in your Swelm application, you can instantiate a `TextField` object and customize its properties using the available methods. Here's an example:

```
TextField myTextField = new TextField(this, "my_text_field")
    .text("Enter text here")
    .placeholder("Type something...");
```

In this example, we create a text field named "my_text_field" with the initial text "Enter text here" and a placeholder "Type something...".

Notes

- TextField provides a way for users to input text into your application.
- It utilizes a custom JPlaceholderTextField component to implement text field functionality with support for placeholder text.

Menu System Widgets

The menu system widgets in Swelm provide a way to create and manage menus and menu items within your application's user interface. These widgets allow you to organize and present functionality in a hierarchical manner, making it easy for users to navigate and interact with your application.

MenuBar

The `MenuBar` widget represents a menu bar component, typically located at the top of the application window. It serves as the container for menus and provides a centralized location for accessing various functionalities offered by the application.

```
public MenuBar(IWidgetCollection widgets, String widgetName)
```

- widgets: An instance of IWidgetCollection interface, providing access to the collection of widgets within the application.
- widgetName: A unique identifier for the menu bar widget.

Main methods

`menus(Menu[] menus)`

Adds menus to the menu bar.

```
public MenuBar menus(Menu[] menus)
```

- menus: An array of Menu objects representing the menus to be added to the menu bar.

Usage

To create a menu bar in your Swelm application, you can instantiate a `MenuBar` object and add menus to it using the `menus()` method. Here's an example:

```
MenuBar menuBar = new MenuBar(thi, "main_menu_bar")

    .menus(new Menu[]{
        new Menu(this, "file_menu").text("File")
            .items(new MenuItem[]{
                new SingleItem(this, "open_item").text("Open"),
                new SingleItem(this, "save_item").text("Save")
            }),

        new Menu(this, "edit_menu").text("Edit")
            .items(new MenuItem[]{
                new SingleItem(this, "cut_item").text("Cut"),
                new SingleItem(this, "copy_item").text("Copy"),
                new SingleItem(this, "paste_item").text("Paste")
            }),

    });
```

MenuItem

The `MenuItem` class is an abstract class representing a single item within a menu. It serves as a base class for specific types of menu items.

```
public MenuItem(IWidgetCollection widgets, String widgetName)
```

- widgets: An instance of IWidgetCollection interface, providing access to the collection of widgets within the application.
- widgetName: A unique identifier for the menu item.

Menu

The `Menu` class represents a menu component within the menu system. It contains a list of menu items and provides a dropdown interface for accessing them.

```
public Menu(IWidgetCollection widgets, String widgetName)
```

- widgets: An instance of IWidgetCollection interface, providing access to the collection of widgets within the application.
- widgetName: A unique identifier for the menu widget.

Main methods

```
text(String text)
```

Sets the text label for the menu.

```
public Menu text(String text)
```

- text: The text to be displayed on the menu.

`items(MenuItem[] items)`

Adds menu items to the menu.

```
public Menu items(MenuItem[] items)
```

- items: An array of MenuItem objects representing the menu items to be added to the menu.

SingleItem

The `SingleItem` class represents a single item within a menu that triggers an action when clicked. It is a subclass of `MenuItem`.

```
public SingleItem(IWidgetCollection widgets, String widgetName)
```

- widgets: An instance of IWidgetCollection interface, providing access to the collection of widgets within the application.
- widgetName: A unique identifier for the menu item.

Main methods

`text(String text)`

Sets the text label for the menu item.

```
public SingleItem text(String text)
```

- text: The text to be displayed on the menu item.

`clicked(ConstructWidget<Msg> app, Msg msg)`

Registers an action listener to handle clicks on the menu item.

```
public <Msg> SingleItem clicked(ConstructWidget<Msg> app, Msg msg)
```

- app: An instance of ConstructWidget representing the application.
- msg: The message to be processed when the menu item is clicked.

Containers

Containers in Swelm are components, which provide different ways of orginizing widgets in an application

BorderContainer

The `BorderContainer` class in Swelm is a widget that represents a container with regions laid out in a border layout. It allows you to organize and manage the layout of your application's user interface by specifying components for each of the five regions: north, south, east, west, and center.

```
public BorderContainer(IWidgetCollection widgets, String widgetName)
```

- widgets: An instance of IWidgetCollection interface, which provides access to the collection of widgets within the application.
- widgetName: A unique identifier for the widget.

Main methods

`north(Widget widget)`

Sets the widget to be displayed in the north region of the container.

```
public BorderContainer north(Widget widget)
```

- widget: The widget to be displayed in the north region.

`south(Widget widget)`

Sets the widget to be displayed in the south region of the container.

```
public BorderContainer south(Widget widget)
```

- widget: The widget to be displayed in the south region.

`east(Widget widget)`

Sets the widget to be displayed in the east region of the container.

```
public BorderContainer east(Widget widget)
```

- widget: The widget to be displayed in the east region.

`west(Widget widget)`

Sets the widget to be displayed in the west region of the container.

```
public BorderContainer west(Widget widget)
```

- widget: The widget to be displayed in the west region.

```
center(Widget widget)
```

Sets the widget to be displayed in the center region of the container.

```
public BorderContainer center(Widget widget)
```

- widget: The widget to be displayed in the center region.

Usage

To create a border container in your Swelm application, you can instantiate a `BorderContainer` object and specify the widgets for each region using the appropriate methods. Here's an example:

```
BorderContainer borderContainer = new BorderContainer(this,
"my_border_container")
    .north(new Button(this, "north_button").text("North"))
    .south(new Button(this, "south_button").text("South"))
    .east(new Button(this, "east_button").text("East"))
    .west(new Button(this, "west_button").text("West"))
    .center(new Button(this, "center_button").text("Center"));
```

In this example, we create a border container with buttons in each region, labeled accordingly.

Notes

- BorderContainer provides a flexible way to organize the layout of your application's user interface using a border layout.
- It utilizes a Container with a BorderLayout manager to manage the layout of its child components.

BoxContainer & Glue

In Swelm, the `BoxContainer` widget is a versatile tool for organizing the layout of user interfaces. It allows developers to arrange components sequentially in either a horizontal or vertical box layout. Additionally, by incorporating `Glue` widgets within a `BoxContainer`, developers can control spacing and alignment between components more precisely.

```
public BoxContainer(IWidgetCollection widgets, String widgetName)
```

- widgets: An instance of IWidgetCollection interface, providing access to the collection of widgets within the application.
- widgetName: A unique identifier for the BoxContainer widget.

Usage

Horizontal Layout with Glue

In a horizontal layout scenario, where components are arranged side by side, horizontal glue can be used to adjust the spacing between components.

```
// Creating a BoxContainer with horizontal layout
BoxContainer horizontalContainer = new BoxContainer(this,
"horizontal_container")
    .align(BoxContainer.BoxAlign.Horizontal)
    .children(new Widget[]{
        new Button(this, "button1").text("Button 1"),
        new Glue(this, "horizontal_glue", Glue.Orientation.Horizontal), //
Horizontal glue for spacing
        new Button(this, "button2").text("Button 2"),
        new Glue(this, "horizontal_glue2", Glue.Orientation.Horizontal), //
Additional horizontal glue
        new Button(this, "button3").text("Button 3")
    });
```

In this example, the `Glue` widgets (`horizontal_glue` and `horizontal_glue2`) provide flexible spacing between the buttons (`button1`, `button2`, and `button3`) within the `horizontalContainer`.

Vertical Layout with Glue

Similarly, in a vertical layout scenario where components are stacked vertically, vertical glue can be utilized to adjust spacing between components.

```
// Creating a BoxContainer with vertical layout
BoxContainer verticalContainer = new BoxContainer(this, "vertical_container")
    .align(BoxContainer.BoxAlign.Vertical)
    .children(new Widget[]{
        new Button(this, "button1").text("Button 1"),
        new Glue(this, "vertical_glue", Glue.Orientation.Vertical), //
Vertical glue for spacing
        new Button(this, "button2").text("Button 2"),
        new Glue(this, "vertical_glue2", Glue.Orientation.Vertical), //
Additional vertical glue
        new Button(this, "button3").text("Button 3")
    });
```

Here, the `Glue` widgets (`vertical_glue` and `vertical_glue2`) create flexible spacing between the buttons (`button1` , `button2` , and `button3`) within the `verticalContainer` .

Customization and Precision

By incorporating `Glue` widgets within a `BoxContainer` , developers gain fine-grained control over the spacing and alignment of components. This approach allows for highly customizable user interfaces that adapt well to various screen sizes and resolutions.

WrapContainer

The `WrapContainer` class in Swelm is a widget that represents a container with components laid out in a wrap layout. It allows you to organize and manage the layout of your application's user interface by arranging components in rows and automatically wrapping them to the next row when necessary.

```
public WrapContainer(IWidgetCollection widgets, String widgetName)
```

- widgets: An instance of IWidgetCollection interface, providing access to the collection of widgets within the application.
- widgetName: A unique identifier for the widget.

Main methods

`children(Widget[] children)`

Adds child widgets to the wrap container.

```
public WrapContainer children(Widget[] children)
```

- children: An array of Widget objects representing the child widgets to be added to the wrap container.

Usage

To create a wrap container in your Swelm application, you can instantiate a `WrapContainer` object and add child widgets using the `children()` method. Here's an example:

```
WrapContainer wrapContainer = new WrapContainer(this, "my_wrap_container")
    .children(new Widget[]{
        new Button(this, "button1").text("Button 1"),
        new Button(this, "button2").text("Button 2"),
        new Button(this, "button3").text("Button 3"),
        new Button(this, "button4").text("Button 4"),
        // Add more widgets as needed
    });
```

In this example, we create a wrap container named "my_wrap_container" and add several buttons to it. The wrap layout automatically arranges the buttons in rows and wraps

them to the next row when necessary to fit within the container's width.

Notes

- WrapContainer provides a convenient way to create responsive layouts with components automatically wrapping to accommodate varying screen sizes.
- It utilizes a Container with a WrapLayout manager to manage the layout of its child components.