Design

Application Design

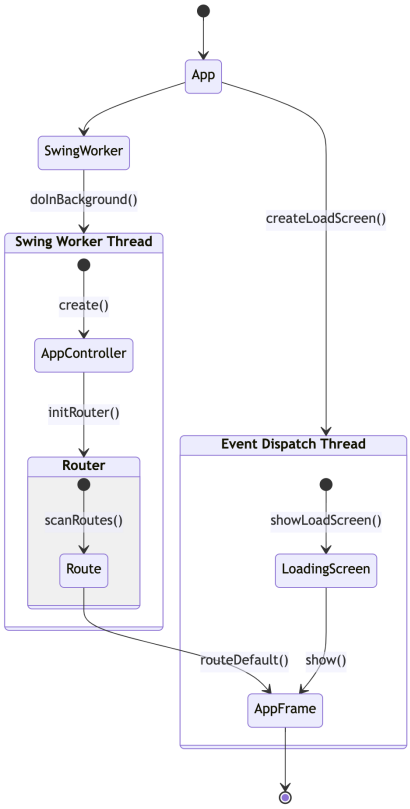
Patterns used

- **Object-Oriented Programming:** Inheritance & Polymorphism
- **Model-View Separation:** MVC Pattern
- **Gang of Four (GoF):** Singleton, Simple Factory, Observer, Strategy
- **GRASP:** Creator, Expert, Controller, Low Coupling, High Cohesion

Object-Oriented Programming

ConKUeror is being developed with the object-oriented programming (OOP) principles in mind. This means that the code is organized into objects that represent the game's entities, such as the game map, the game deck, and the player objects. OOP is also aligned with our “Modularity” principle, which means that the code is easier to read, maintain, and extend.

Some common OOP patterns used in the development of **ConKUeror** include **inheritance** and **polymorphism**. Inheritance is used to create subclasses of the army objects, such as the Infantry, Cavalry, and Artillery objects. **Polymorphism** is used to allow the game to handle different types of objects in a consistent way, such as when determining the outcome of an attack. Use of OOP principles allows for a more organized and efficient development process, resulting in a better game for the end user.



Diagrammatic representation of the **application startup** process.

Application Startup

Builder

The `AppController` and `Router` classes use the Builder Pattern to construct the ConKUeror application in

steps, as shown in the diagrammatic representation of the ConKUeror startup. This allows for efficient and flexible construction of complex objects, which enhances the application's performance and responsiveness.

See:

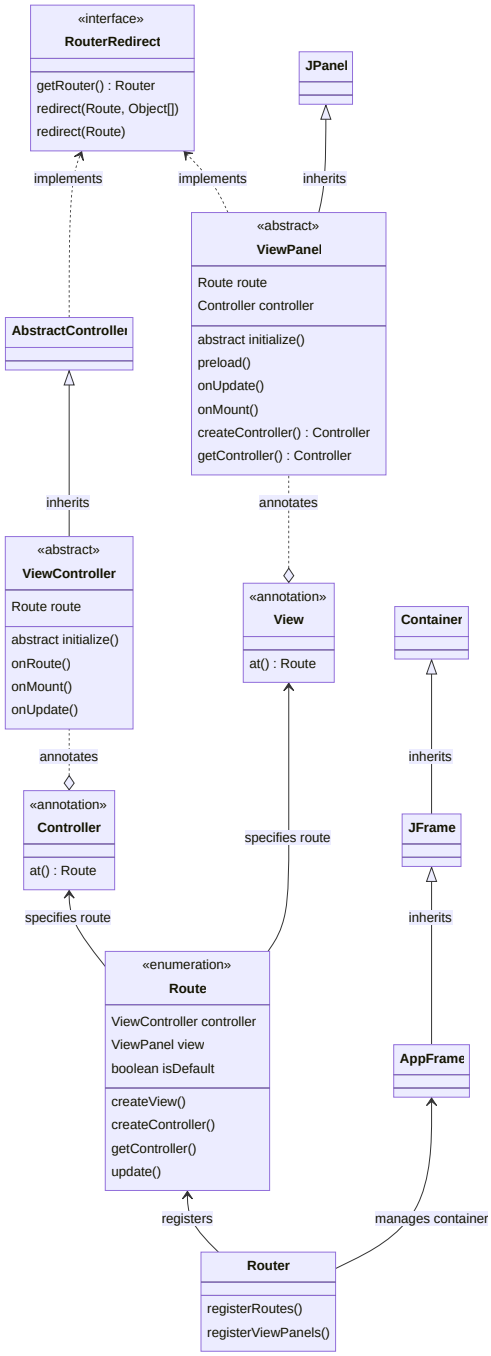
- `conkueror.ui.app.frame.AppController`
- `conkueror.ui.app.frame.LoadingScreen`

Creator

The `Router` class in ConKUeror uses the Creator pattern to create and cache `ViewPanel` and `ViewController` objects. The `Router` reads the annotations on each `ViewPanel` and `ViewController` to determine their associated route, and uses the `initialize()` method on each object to instantiate it. Once the object is created, the `Router` adds it to a cache for later use.

This approach allows for dynamic creation and caching of `ViewPanel` and `ViewController` objects, which can be useful in situations where large numbers of objects need to be created and managed. By using the `initialize()` method to create the objects, the `Router` can ensure that each object is properly configured before it is used, and by caching the objects, the `Router` can reduce the overhead associated with creating new objects.

The use of annotations to associate each `ViewPanel` and `ViewController` with specific routes also provides a convenient and flexible way to manage application navigation, as it allows developers to easily add, remove, or modify navigation routes without having to modify the individual codes.



Class diagram for the **Router** and its relation to the navigation system.

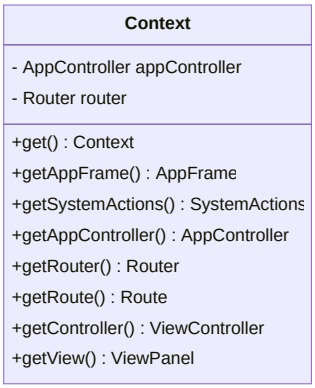
Multithreading

In ConKUeror, the loading of necessary resources and initialization tasks are performed using the **Swing Worker Thread**, by invoking the `preload()` method of each ViewController, which allows the loading process to happen in the background while the user sees the loading screen. This ensures that the main thread is not blocked and that the application remains responsive. The Swing Worker Thread also provides

a built-in mechanism for canceling the loading process and handling errors. By using the Swing Worker Thread, ConKUeror is able to provide a fast and responsive interface while still ensuring that important resources are loaded before the main screen appears.

See:

- `conkueror.ui.app.App`
- `conkueror.ui.app.frame.AppFrame`
- `conkueror.ui.app.frame.LoadingScreen`



Global Application Context

UI Rendering

Model-View Separation

In ConKUeror, model-view separation is achieved by separating the model, user interface, application code, and controller classes into different packages. The model is located in the `domain` package, while the user interface, application code, and controller classes are located in the `ui` package. This separation allows for a clear separation of concerns and makes it easier to maintain and modify the code.

Controller

In ConKUeror, the ViewController pattern is used to handle communication between the domain and the UI. ViewController classes are responsible for handling user input and updating the UI accordingly. ViewPanels are responsible for rendering the Swing components, and have their respective ViewController as a type parameter.
For example, `MainView extends ViewPanel<MainController>`. This allows for easy access to the controller's methods throughout the view panel, by calling the `getController()` method. This approach follows the **Facade pattern**, which simplifies the interface to a complex system by providing a unified interface to a set of interfaces in a subsystem.

- `conkueror.ui.app.router.ViewPanel`
- `conkueror.ui.app.router.ViewController`

Multithreading

Multithreading is also used in ConKUeror to handle animations and other time-sensitive operations, ensuring that the UI remains responsive and fluid even during complex or resource-intensive tasks.

Application Navigation

Facade

In ConKUeror, the custom router class, `Router`, for navigating and constructing view panels and view controllers uses the Facade pattern implemented in the `ViewController` class. This class implements the RouterRedirect interface, which defines the `redirect(Route route)` method for navigation. By encapsulating the logic for constructing the view panel and view controller for the target view using the Facade pattern, the router provides a simplified interface for navigation while reducing coupling and hiding irrelevant implementation details from UI code.

The facade pattern is also used to simplify the interface of the ViewPanel and ViewController lifecycle hooks. These hooks are implemented as separate methods with specific names, such as `preload()`, `initialize()`, `onRoute()`, `onMount()`, and `onUpdate()`. However, they are all activated on the chain of responsibility by the same facade method, `redirect()`, which is used for navigation.

When the application is loading, it first calls the `preload()` method, which is used to load any necessary data or resources before the view is initialized. Then, it calls the `initialize()` method, which is used to set up the initial state of the view and controller. Finally, it calls the `onRoute()`, `onMount()`, and `onUpdate()` methods as needed, depending on the specific lifecycle of the view.

By using the facade pattern, ConKUeror is able to provide a simple and consistent interface for managing the lifecycle of its views and controllers, while still allowing for a high degree of flexibility and customization. This makes it easy to add new views and controllers to the application, and to modify or extend the behavior of existing ones.

See:

- `conkueror.ui.app.router.Route`

Decorator

In ConKUeror, the decorator pattern is implemented using annotations. ViewPanels and ViewControllers are annotated with `@View` and `@Controller`, respectively, and provided with the specific route they handle, for example, `@Controller(at = Route.Main)`. This enables the separation of concerns between different parts of the application navigation, while still allowing for the addition of behavior to individual objects in a flexible and dynamic way.

See:

- `conkueror.ui.app.router.View`
- `conkueror.ui.app.router.Controller`

Chain of Responsibility

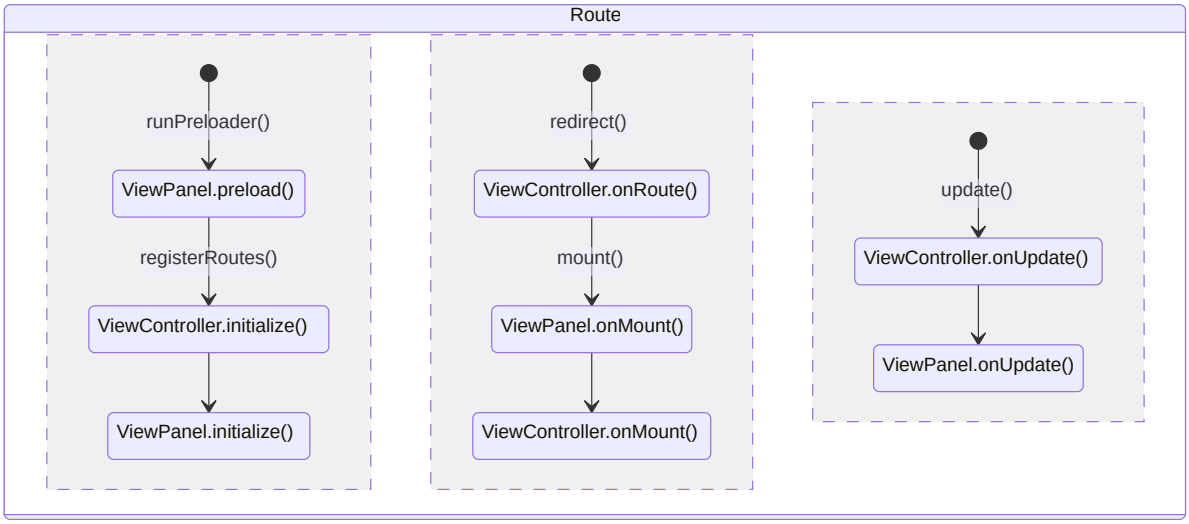
In ConKUeror, the router class uses the `redirect(Route route, Object... args)` method to navigate between views by propagating the route through a chain of responsibility. The method takes a `Route` object and an optional list of arguments, which can be used for passing data between views.

By using the chain of responsibility pattern, the router class provides a flexible and extensible system for navigation between views. New `ViewController` objects can be added or removed easily without affecting the existing ones, and different `ViewController` objects can be used for different types of events.

The chain of responsibility design pattern can be particularly useful for handling UI updating and rerendering in ConKUeror. When an update is triggered from anywhere in the application, including the global application `Context` object (a singleton), the update signal is handled on the chain of responsibility and delegated to all the necessary application objects, such as views and controllers. This ensures that the update is propagated throughout the application in an efficient and organized manner.

See:

- `conkueror.ui.app.router.AbstractController`
- `conkueror.ui.app.router.Route`
- `conkueror.ui.app.Context`



Route lifecycle. Left: Application Startup. Center: Application Navigation. Right: View Updates

Strategy

In ConKUeror, the strategy pattern is implemented using the view and controller hooks. Views and controllers are the two fundamental components of the ConKUeror application design that handle the communication between the domain and the UI. The hooks are used to define the different stages in the lifecycle of a view or controller.

Hooks are triggered in three events: *Startup, Navigation, Update.*

Views have four hooks:

- `preload()`: This hook is triggered during the startup phase of the application. Each view is responsible for preloading their resources using this method. This allows the view to load any necessary data or resources that will be needed before the view is initialized.
- `initialize()`: This hook is responsible for initializing the view. It is triggered during the startup phase of the application to set up the initial state of the view.
- `onMount()`: This hook is triggered when the view is mounted onto the UI. It is responsible for rendering the view and performing any necessary operations. This method is used to ensure that the view is properly rendered and ready to be displayed.
- `onUpdate()`: This hook is triggered when an update event occurs. It is responsible for updating the view with the new data.

Controllers also have four hooks:

- `initialize()`: This hook is responsible for initializing the controller. It is triggered during the startup phase of the application to set up the initial state of the controller.
- `onRoute(Object... args)`: This hook is triggered when a navigation event occurs. It accepts arguments using the `Object... args` parameter and can be used to do work before the navigation takes place. This method is used to ensure that the controller is properly configured before it is used.
- `onMount()`: This hook is triggered when the view is mounted onto the UI. It is responsible for performing any necessary operations on the view. This method is used to ensure that the

controller is properly mounted onto the view.

- `onUpdate()`: This hook is triggered when an update event occurs. It is responsible for updating the view with the new data.

By using hooks, **ConKUeror** allows for the implementation of different strategies for the same component without having to modify the component's code directly. This means that a view or controller can be reused across different contexts with different strategies applied to it.

Adapter

The adapter pattern in **ConKUeror** is used to provide a bridge between the game's domain model and the save/load functionality. The adapter pattern allows the use of an existing interface to work with a new one, without having to modify the existing interface. In this case, the domain model is the existing interface, and the save/load functionality is the new one. The adapter, in this case, would implement the save/load functionality and use the domain model to perform the necessary operations to save and load the game state. This allows the game state to be saved and loaded without having to modify the domain model.

Design Principles

Ease of Use

The player needs an easy-to-use UI and intuitive controls to interact with game objects. Keep in mind that **ConKUeror** is a computer game based on the board game RISK and, by design, is a strictly game-controlled environment. Because the game has many moving parts, a simple help page is not sufficient for users to understand it. Therefore, intentional use of attention-grabbing icons is necessary to represent objects such as the game deck, cards, chance card effects, and so on, as well as a straightforward user panel that displays all possible actions, wherever applicable and relevant. This is a principle that we actively utilize in designing the game.

The infamous cover of the book, Structure and Interpretation of Computer Programs, but programmers are receiving messages from God — edited by AI.

Adaptability

Creating a captivating game environment requires effective system messages and announcements. For instance, when the game presents a chance card, the screen blurs out to focus the player's attention on the card's content. The message also includes big, relevant buttons for that chance card and a "skip" option. The panel always displays relevant information based on the current game setting and phase. During a player's turn, the panel shows easy-to-see buttons to exchange armies or use army and territory cards, but only if the player satisfies the requirements for doing so. This ensures that the user sees only the options they can use at that moment, rather than all of the game options at once.

Configurability

The game can adapt to different numbers of players and user preferences. The game map can be modified according to the players' needs. Additionally, depending on the number of players, ConKUeror is designed to utilize different rule sets to provide a fair experience for all players.

God is pondering whilst writing his first computer program — generated by AI.

Reliability

Like any other application, ConKUeror is not immune to bugs and errors. If an error occurs, the system should display a message on the screen that includes the error, as well as a button to send the error to the developer. Additionally, the game should be saved and restarted to prevent the loss of user data.

Performance

As mentioned, the game should be built using one of the most potent and high-performing languages available, utilizing a fast and flexible core library to make the game run smoothly even on low-end machines. Performance is also a priority for the user interface and animations. As a 2D board game, ConKUeror utilizes the reliable Java library Swing to implement all custom UI objects, animations, and user interactions, minimizing the risk of unnecessary resource usage.

Game map with noticeably distinct colors for visual aid — generated by AI.

Cross-Platform Design

Java's cross-platform capabilities, made possible by its support for JVM, allow it to run on any operating system. This is particularly helpful, as it would enable the development of a network gaming option in the future. Players with different machines and operating systems could play together and enjoy the same gaming experience.

Modularity

The game should be packaged in a way that makes it easy to extend and customize. This would allow for easy additions of further chance cards and army types, new countries and continents, or even different game maps and themes.

Constraints & Limitations

ConKUeror uses only Java standard libraries and Swing. The custom UI components are also designed using these same libraries.

Technical Glossary

- **Java**: A popular programming language used for developing a wide range of applications.

- **JVM**: Java Virtual Machine, a virtual machine that executes Java bytecode.
- **Java Standard Libraries**: A set of libraries included with Java that provide a wide range of functionality.
- **Java Swing**: A GUI toolkit for Java used for developing desktop applications.
- **Class**: A blueprint or template for creating objects.
- **Object**: An instance of a class.
- **OOP**: Object-Oriented Programming, a programming paradigm that focuses on objects and their interactions.
- **Model-View separation**: A design pattern that separates the data from the presentation of the data.
- **MVC**: Model-View-Controller, a design pattern used in software engineering that separates the application into three interconnected components: the model, the view, and the controller.
- **Model**: The part of the MVC pattern that represents the application's data and business logic.
- **View**: The part of the MVC pattern that represents the presentation of the data to the user.
- **Controller**: The part of the MVC pattern that handles user input and interacts with the model and the view.
- **Domain Model**: A conceptual model that represents the real-world entities and their relationships in a particular domain.
- **System:** the **ConKUeror** game and its underlying architecture and components, including the domain model, user interface, and game logic.
- **User:** A user is an individual who interacts with a system, such as a computer program, website, or application. In the context of this document, a user would refer to someone playing or interacting with the **ConKUeror** game.