Design

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.

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

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.

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.

To ensure security and reliability, the game should be built using one of the most popular and advanced programming languages in the development world: Java. This makes it highly reliable and secure, allows for system and

architecture-independent game development, and eliminates unexpected behavior.

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

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.

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.

Implementation Principles / Patterns

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.

Model-View Separation

In the development of **ConKUeror**, we utilize the Model-View-Controller (MVC) design pattern. This pattern separates the application logic into three interconnected components: the model, the view, and the controller.

- **Domain Model: Model + Controller**
  Our model and controller are combined in one layer we call the `Domain`, as specified in the section with the same name. All the underlying data and game logic — the model — are implemented in game objects, while the controller acts as the intermediary between the model and the view. App manages game flow, player turns, game state transitions, and so on, actively utilizing the **Observer Pattern**, as specified above.
- **User Interface: View**
  The view is the UI abstraction layer, as specified in the next section, responsible for rendering the model data to the user interface, including components such as the world map, the cards, armies and interactive components like the buttons and inputs.

By separating the model from the view and the controller, we have created a modular and flexible design that allows for easier maintenance and extension of the game.

Controller

The controller pattern is actively used in **ConKUeror** to manage the application and game state. Both the `App` and the `Game` classes act as an intermediary between the model and the view, responsible for managing the game flow and providing the necessary data to the UI components. The `Game` class actively utilizes the **Observer Pattern** to ensure that the UI is updated whenever the game state changes.

The `Game` class also manages the various game components, such as the game deck, the game map, and the player objects. It ensures that these components are updated and managed correctly, and that the game logic is executed correctly.

Expert Pattern

The expert pattern is a design pattern used in software engineering that assigns responsibilities to the objects that have the necessary information to fulfill them. In **ConKUeror**, the controller and the specific `Game` instance uses the expert pattern to delegate responsibilities to the game objects that have the necessary information to perform specific tasks.

For example, when a player attacks an enemy territory, the game needs to determine the outcome of an attack and update the state. Instead of having the `Game` class handle all of these factors, the responsibility is instead delegated to the `Round` class, which in turn uses the `Dice` and `Deck`, which have all of the necessary information to determine the outcome of the attack.

Singleton

Singleton is a design pattern that ensures only one instance of a class can be created and provides a global point of access to that instance. This pattern is actively used in **ConKUeror** to ensure that there is only one instance of a class that manages the game state and flow, also allowing this class to be accessed by all other classes that require this information.

This is especially useful since it ensures a single instance of a class, which can be useful for managing global resources and state, and provides a global point of access to that instance, which simplifies the code and makes it easier to manage dependencies.

However, using the Singleton pattern alone is not thread-safe, which means that if multiple threads try to access the Singleton instance simultaneously, it can lead to race conditions and other issues. To address this, **ConKUeror** uses the **Bill Pugh Singleton pattern**, also known as the initialization-on-demand holder idiom, which is thread-safe and ensures that only one instance of a class is created even in a **multi-threaded environment**.

Simple Factory

The **Simple Factory pattern** is used to create the game map in **ConKUeror**. For example, the `DefaultWorldMap` class uses `createMapItem()` in reference to its adjacency matrix to create the different territories and continents for the map.

Creator Pattern

The Creator Pattern is a design pattern used in software engineering that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. In **ConKUeror**, this pattern is used to create army objects. For example, the `ArmyCreator` interface defines a method called `createArmy()` that returns an `Army` object, and the `InfantryCreator` and `CavalryCreator` classes implement this interface to create infantry and cavalry armies, respectively. This allows for a more flexible and modular design, as new army types can be easily added by creating new classes that implement the `ArmyCreator` interface.

For example, chance cards utilize this pattern to add a new type of armies called the `Mercenary` using the `MercenaryCreator`.

Observer Pattern

The Observer pattern is a design pattern used in software engineering that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. In **ConKUeror**, the Observer pattern is used to ensure that the game UI is updated to reflect changes in the game state.

For example, when a player makes a move, the game state changes, and the UI needs to be updated to reflect this change. By using the **Observer pattern**, the game is designed so that the UI is automatically updated whenever the game state changes. The main advantage of the Observer pattern is that **it decouples the subject** (the object being observed) from its observers. This means that changes to the subject do not affect the observers directly, but instead, the observers are notified of the changes and can update themselves accordingly.

However, a large issue with the Observer pattern is that it can lead to a large number of updates being sent to observers, which can be inefficient and lead to performance problems. Additionally, if the subject and observers are tightly coupled, it can be difficult to maintain and extend the system.

To address these, **ConKUeror** sends **updates locally and on-demand**, and utilizes **Low Coupling** and **Strategy.** For example, when a player moves an army from one territory to another, the game only sends an update to the UI for the affected territory, rather than updating the entire game map. Additionally, **ConKUeror** uses the Observer pattern in combination with other design patterns to create a more efficient and modular system. For example, the game uses the **Strategy Pattern** to determine how to handle different types of game events. This allows for more efficient and targeted updates to the game state, reducing the number of updates that need to be sent to observers.

Low Coupling

ConKUeror defines clear interfaces and uses dependency injection to provide the necessary objects to each class, actively utilizing the **Low Coupling pattern**.

For example, the `Game` class defines an interface for the `WorldMap` object, which is implemented by the `DefaultGameMap` class. The `Game` class is then instantiated with an instance of the `GameMap` interface, without knowing which concrete implementation is being used. This ensures that we can at some point add a new game map to the game for variety.

Similarly, the `Player` class defines an interface for the `Army` object, which is implemented by the `Infantry`, `Cavalry`, and `Artillery` classes. This allows for new army types to be added without affecting the `Player` class. `Mercenary` is an example for this.

High Cohesion

In **ConKUeror**, high cohesion is achieved by organizing classes into packages based on their functionality. For example, all classes related to the game map are located in the `game.map` package, while all classes related to the game deck are located in the `game.deck` package. Each package has a well-defined purpose, and all classes within that package are related to that purpose.

Additionally, each class has a single, well-defined purpose. For example, the `Game` class is responsible for managing the game state and flow, while the `Player` class is responsible for managing the player's armies and territories.

To document the purpose of each class and package, we are using java decorators such as `@Category`, `@Description`, `@Label` and IntelliJ decorators such as `@NotNull` and `@Nullable` etc., for annotations.

Strategy

The Strategy pattern is a design pattern that allows the behavior of a class to be changed at runtime by providing different implementations of a particular algorithm.

For example, the `ChanceCard` class defines a `ChanceCardEffect` interface that represents the effect of the chance card, and the `ChanceCard` class has a `setChanceCardEffect()` method that allows the effect to be set at runtime. Different chance cards are then created by implementing the `ChanceCardEffect` interface with different behavior. For example, the chance card *Bombardment* implements the `ChanceCardEffect` interface with a `rollTwo()` method, while *Reinforcements* allows the player to move armies by implementing the `ChanceCardEffect` interface with a `moveArmies()` method.

The `ChanceCard` class also uses the **Singleton pattern** to ensure that there is only one instance of the `ChanceCardEffect` object for each type of chance card.

Strategy pattern also allows us to define different game rules for different stages. `Game` class uses a `GameRules` interface to define the rules for the game, which is implemented by the `GameInitRules` and `GameTurnRules` classes. This allows for a seamless transition between the initial sharing of territories stage to the rounds stage.

The `ChanceCard` class in **ConKUeror** uses the **Strategy pattern** to allow different chance cards to modify the rules of the game. The `ChanceCard` class defines a `ChanceCardEffect` interface that represents the effect of the chance card, and the `ChanceCard` class has a `setChanceCardEffect()` method that allows the effect to be set at runtime.

Adapter Pattern

There is no necessity to use this pattern, unless we decide to support a network gaming option.

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.