Department of Computer Engineering

Bilkent University

CS319 Term Project

Section 1

Group 1D - Risk Takers: Risk Board Game

Design Report

1. Esad Burak ALTINYAZAR
2. Burak YENI
3. Burak MUTLU
4. Yigit Kutay GULBEN
5. Nurlan FARZALIYEV
6. Anar HUSEYNOV

Supervisor: Eray TUZUN

Contents

1. Introduction

1.1 Purpose of the System:

The purpose of implementing digital version of Risk box game can be thought as playing such a fun game on a digital platform so that playing the game would be much quicker and additional features may be added into game, which makes it even more fun. The reason for being much quicker experience is that you do not need to setup physical conditions to play game with friends. By implementing additional features, we aim to offer a better quality and unique experience for users. That is, the purpose of the system is introducing innovative and quality features, in which main concerns are usability, robustness, extensibility and performance.

1.2 Design Goals

1.2.1. Trade-Offs

1.2.1.1 Memory-Performance Trade-Off:

As it is an issue in all types of programs, we have faced the same question that is related to memory and time complexity. Therefore, we can state the fact that Memory - Performance problem was the hardest trade-off that cannot be ignored. On the other hand, it was clear that the performance was more important than memory in our trade. Mostly in games which require to run highly complex graphical engines also bring huge data with them, and that increases the memory size in the program. Considering 2D graphics, there is not huge-sized data stored in memory to provide the data for program runs over. We are simply loading the visual and model data to the memory while initializing the game and trading off the memory, we got better performance while game running. That is, the aim in that manner was increasing our performance in order to keep the game simple and fun for the user aspect. Only trading off memory over performance is that it does not load the game files if application is in the state of menu. Then, while initializing game, it requires a few seconds to load those data, which actually is not efficient for performance but better for memory in menu state.

1.2.1.1 Graphics Quality-Performance Trade-Off:

As mentioned, we are using 2D graphics; in advance we actually much more trading off performance over graphics quality since also pixel theme included in game design. Pixel theme decreases the quality of graphics in appreciable level depending on which scale you pixelized the visuals. We may say that our pixelization factor is in a intermediate level, in which you may get enough performance increase and not giving up required graphics quality to saturate the players' game experience. In a sense, while using pixel theme even if you are giving up graphics quality, the game experience is increased because of not only better performance but also an offered game-theme. Using game-theme term, we mean involving

animations, sound effects and gameplay which follow common design style of theme and as a combination, it will resulted in eye-catching game design and experience. On the other hand, implementing animations while using the pixel-theme visualization is much easier and enables us to come up with much more dynamic and complex animations. For these respects, we are trading off performance and game experience over graphics quality.

1.2.2. Criteria

1.2.2.1 Performance Criteria:

Game Performance: In essence of a game, the users are the main focus of games, they are the target community of implemented program. Since the reflected performance to the users affects the experience they get, game performance is an important consideration. That's why we should focus to optimize the viewed performance by users for offering much flowing game experience. In such issue, most obvious subsystems which performance consideration taken into account is ArtificialIntelligence, SoundEffects, Animations. While optimizing those, Artifical Intelligence should performs reasonable moves and seems like it has advanced tactics to play against components. Sound Effects and Animations should feel the player enjoyable to play such game, that is, there should be some association and hierarchy between those dynamically performing instances. So, we should optimize these type of relations and as a result, it just feels that the game has a flowing experience.

Efficiency: While implementing program especially game-like programs which highly demands on the process power, the efficiency of memory and time complexity of algorithms should be taken into account. For this respect, we regard the time complexity of algorithms used to simulate the game mechanics since they may have high effect on the game performance. Also, memory related issues may affect the execution of game functionalities in long term. For the sake of having good performance, such factors should be always in first priority among the implementation of game. In such perspective, we came up with some data structures which are appropriate for some objects. For example, we used graph data-structure to simulate the connections of territories with each other. Using graph, it provides much efficiency while trying to detect whether units are transferable from one territory to far another during the fortify phase of game. On the other hand, we used double-array structured visual buffer for animation effects to enable direct access specific row and column indexed elements and manipulate them for animation purpose. In other respects, we sometimes divided the lists of same classes' instance into sublists with the purpose of accessing them much quicker since they are related with different mechanisms.

1.2.2.2. Dependability Criteria:

Reliability: The last thing a user want to experience is the crash of a game in the middle of the game. In the matter of whole offered options in the game, there should not be any bug in its complete boundaries, and, the game should fully perform its options. The structure of our program and game mechanisms is designed to not allow any undesired situations. For other respects, in game mechanics, we should not leave any open-end relation such that a smart user may realize it and try to abuse over such open-end relation. That would be a very annoying experience for the players, especially in multiplayer game mode.

Robustness: Our game will have a solid stand in the view of users through its additional features and well thoughts characteristics. For instance, the mentioned well thoughts characteristics of the game may be explained as its fascinating visual effects and animations, which are supported with harmonic sound effects, its addictive game experience, offering a competitive multiplayer or strategically deep-minded computer opponents.

Playability: Our game will be understandable for the players as their first look into game; we will offer supportive hints for most essential rules and mechanisms of the game and also a tutorial game mode for beginners to get adapted with the game. As an additional feature, we may implement a specific feature like in-game dynamic difficulty, which refers to changing difficulty into much easier or much harder ones dependently on the player score gap with others.

1.2.2.3. Maintenance Criteria:

Usability: The “Risk” game is a board game originally. What is problematic about the board games is those kinds of games require some other players to be present at a time in order to play the game, and managing the game itself can also take some time. However, the usability of the version that we will be implementing will be very high since it will take just a couple of seconds to open the game and start playing. Moreover, there is no need for other players to be present, there will be “Single player” option as well.

Extensibility: It should be always exciting to update the game with additional features after implementing of its primal version. That’s why we will always be ready for further improvements and modifications and we will realize such game implementation that it would be available to integrate with implementations of upcoming features. Since we are using Java to implement the game, our design is going to be an object-oriented software design. This design ensures that by future it will be easy, will not be problematic, to make changes or add new features in case of any problems or enhancements. For example, we are planning to have one default difficulty level before sum up the whole implementation. Afterwards, we will add new difficulty levels as well, and it will not take too much time.

Adaptability: The main reason we choose to implement our game in Java is it provides platform independency. The compiled Java code can be run in any platform that runs JVM.

2. System Architecture

2.1 Subsystem decomposition

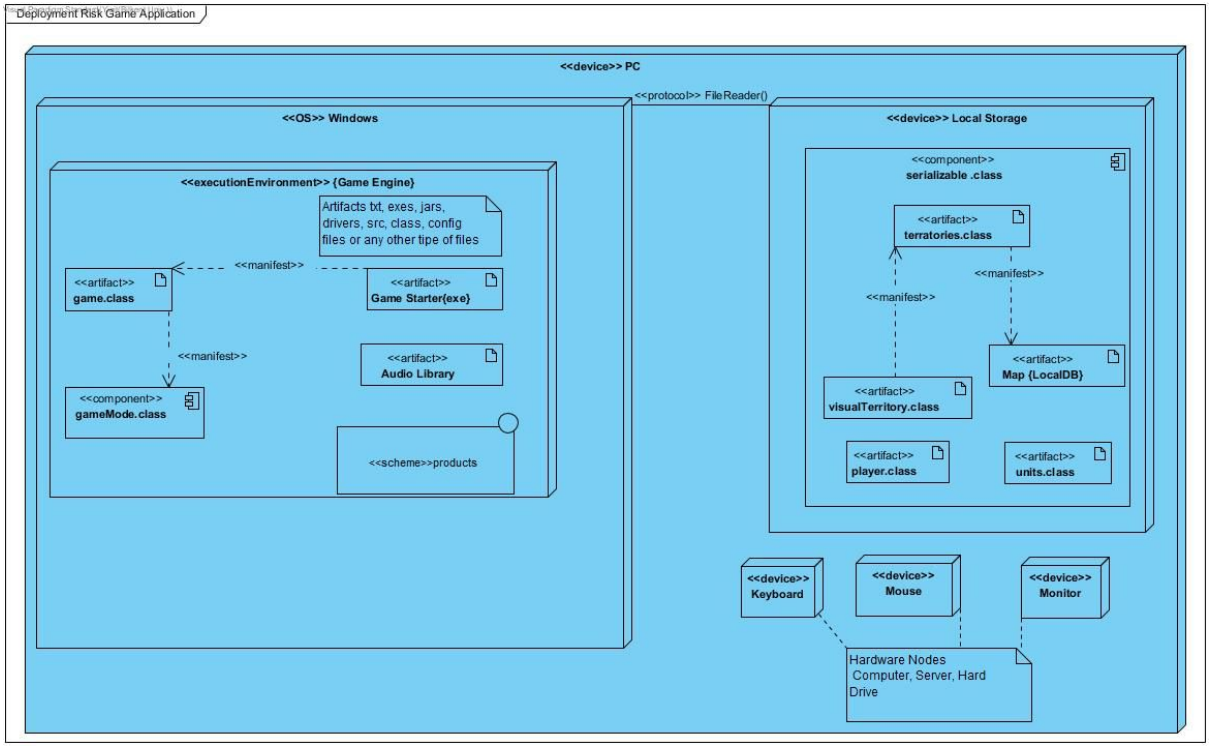
While implementing a game, the most basic way to make complex intertwined things much easier to handle is dividing the functionalities into subcomponents. For such purpose, we choose to implement **MVC** (Model-View-Controller) programming approach. By using MVC, we will divide the complex game functionalities into such three segments that all three executes its own responsible stuff and communicates with each other properly to give a program running without any sort of error.

Model: Model describes the structure of data and business logic. Objects of model retrieve and maintain the state of the model in a program. Applies the game mechanics into their current state and provides the required data to the other components, which are View and Controller.

View: View is also known as user interface. In its most basic sense, it describes the visual reflections of the corresponding model objects. The interface not only displays the model to the user and but also makes the data modifiable since it is related with user inputs. It will transmit the inputs not directly into model components but the controller. That's a better way of implementing things.

Controller: Controller manages the user request and executes required operations over model components. Generally, user firstly interacts with the View and after that needed URL request will be generated by this component, that is, generated request will be fulfilled by a controller.

2.2. Hardware/Software Mapping



Deployment Diagram of our application above shows the interactions between hardware and software, in order to represent both sides of the communications. Major interactions initialized by the player itself through the software triggered by the hardware system is starting from the executable environment which is the Game Engine that is located in the Hard Disk that also includes the Local DataBase of the game. DataBase is having all kinds of information that is required to play the game for different individuals as well as in the single player mode AI is there also. "<<scheme>> is the products that are actually objects

created by the game engine such as player objects, unit objects, territory objects and many more. Artifacts are the classes, exes, configs, jars, txts, srcs or any types of files other than the physical components. Since the our version of the Risk Game is only using the localDB, datas are stored in the Hard Disk and therefore <<protocol>> that is transferring the data from Data Base to Game is just FileReader().

2.3 Persistent Data Management

There will be background photos in the game, music will be one of the features. The files of such features may be stored in local storage. The all required game data needed to be saved like game instances, objects and all of these kinds of data will be saved in an ordinary file. Since the persistent data is not too much, we have decided to save our game as a database stored in local memory. The saved stuff will be encrypted and user will not have access to the saved files.

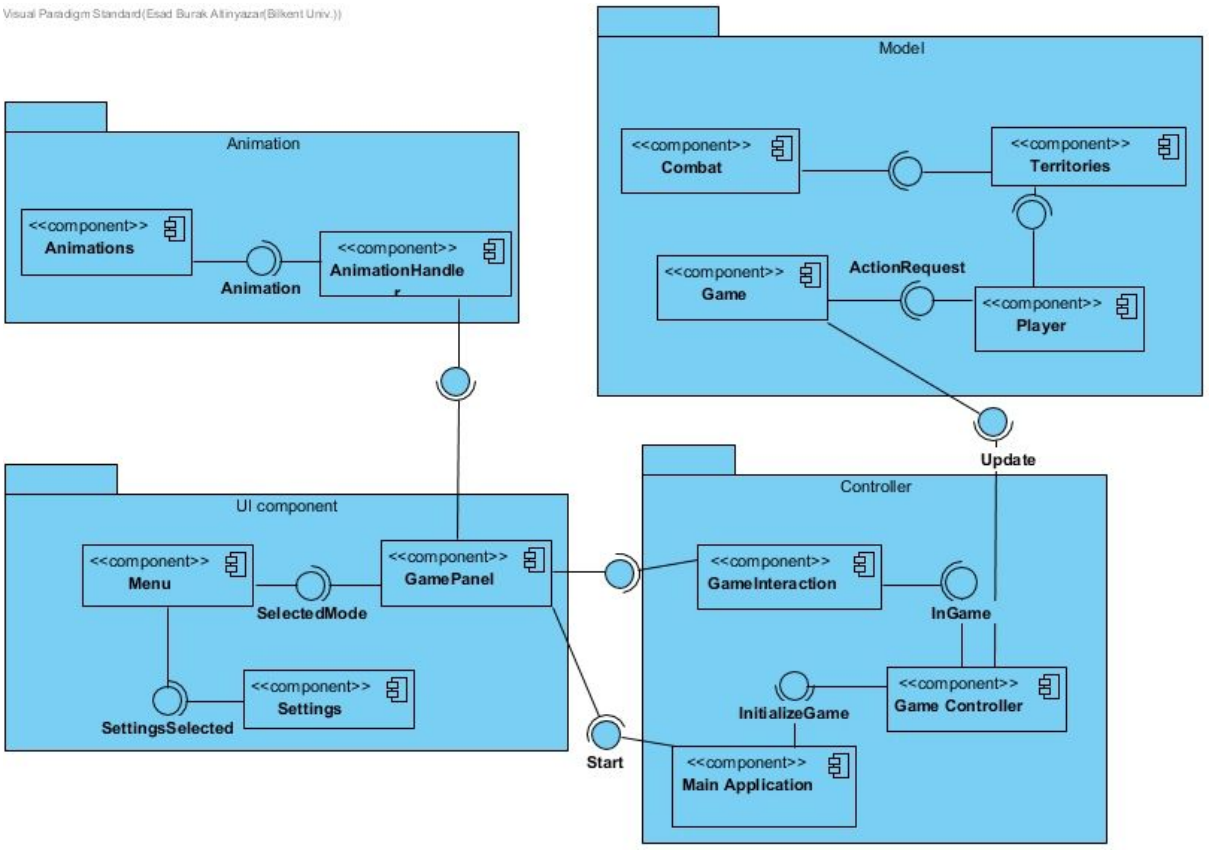
2.4 Access Control and Security

In order to make the data secure we decided to encrypt the saved files and those files will not be user accessible. The encryption always makes things safer even if it is not needed. The network handles the requests and corresponding operations of such requests, which resulted in that the user not able to manipulate the game data directly. For such manner, the user is able to only communicate with networks that is supposed to execute the plausible requests of users, that is, it is a middle layer restricts the user access and provides better security.

2.5 Boundary Conditions

It does not matter in which perspective you look, the games either offer to the players some freedom and determines some restrictions over their options. We need to set such boundaries on the options game offered in order to balance game experience and not to lead any boundary abusing. If any kind of error occurs during the game the program will exit. Player cannot run the game while it is already on run, so first application will exit automatically if the player reruns the game. Through the System library in Java, we will detect the application is already running and take required actions to not give any abuse possibility.

3. Subsystem Services



System is divided into 4 subsystems UIComponents,Controller,Animation,Model. UIcomponents contains menu settings and gamepanel components.Menu provides selection that can be setting or gamePanel. GamePanel holds the graphic properties of the game.Therefore, GamePanel Component has a relations with AnimationHandler MainApplication and GameInteractions.

GameContoller handles the model property changes and send the changing request to the game component in model class. As the request send to the Game in model class by actionrequest properties of the player changes as the properties of the player, properties of the territories also changes.

As the changes occur in game animation requests are called from animationHandler and animations are executed by animationHandler until they are terminated.

3.1 User Interface Subsystem

UI Manager, GamePanel is the base classes for all the considered input, they are responsible for all the territory selections, attacking, adding unit and so on. They take the input from the player through the mouse and buttons, and passes that information to the GameManager and SettingManager.

This View Classes monitors the game to the users with respect to Input Managers and the model classes. The Game will react to the users input which are taken and using this input

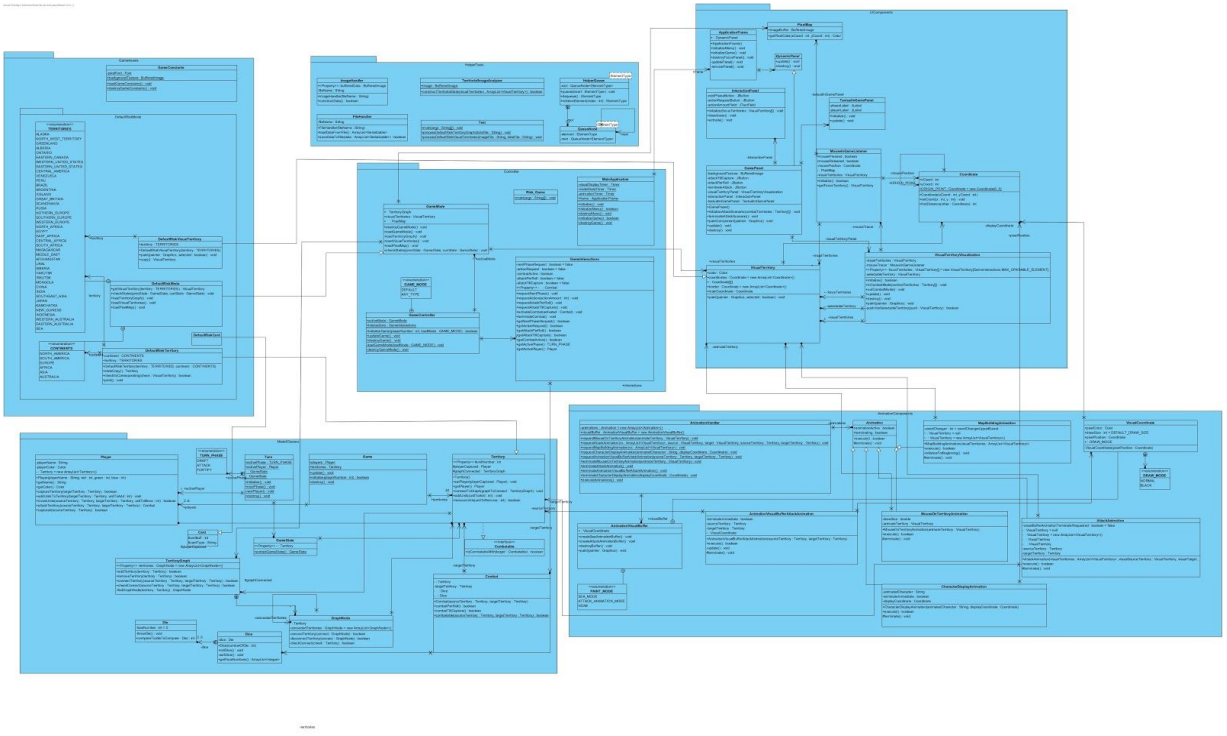
games continues by changing the model's properties. And monitors the model classes and reflect the game to the screen.

3.2 Model Subsystem

Model classes can be listed as Game, Turn, GameState which are top-level running game classes. Basically, they will determine which map is used, in which player turn is and current phase of the game. Much lower classes determines the cards needed to be given players after a turn, the moves a player is able to do and how combat mechanics works.

4. Low Level Design

4.1 Final Object Design



[Complete Class Diagram Link (In case of unreadibility)]

Here, we have the illustration of final object design. Using MVC, we have Model, UI Components and Controller packages for essential functionalities. We extended the application with Animation packages, which defines the hierarchy of animations instances with other animations or some visual classes. On the other side, we have Game Assests to define game-mode specific features, unique characteristic of game-map. Also, HelperTools package involves classes and tools developer to use and make easy to develop the game.

4.2 Design Decisions - Design Patterns

Certain design pattern are used in the implementation to make our system composed of more consistent components. That is, after getting used to the reasoning of proposed design patterns, integrating the components and subsystems was much easier to implement and there were more rapid development phase in later. In this section there will be explanation about what kind of design patterns are involved in our implementation and why we used them.

4.2.1 Creational Design Patterns

4.2.1.1 Abstract Factory Design Pattern

Abstract Factory pattern is much reasonable to use when extensibility and improvability is main concern. In respect of such concerned requirements, we mean the adding new features to the game while we should not need to modify already existing implementation. Abstract class provides such a convenience through already constructed relations between other type of instances of application. And these relations also implies in same manner for recently extended child class of those abstract class. Using Abstract Factory pattern, we had GameMode, Territory, VisualTerritroy, Card abstract classes to implement essential behaviour that implies same for all specialized types. Using those abstract references, we constructed a common relation with other objects and then just extending such classes with DefaultRiskMode, DefaultRiskTerritory, DefaultRiskVisualTerritory, DefaultRiskCard to give unique characteristics to the instances of those class. Clarifying mentioned unique characteristics, for example, we mean making instances belongs to a specific territory-namings (Alaska or Madagascar etc.) and specifying the buff amount of cards associated with the territory-namings, that is, giving them their game-map and mode specific attributes. For such manner, we abstracted the common and essential functionality from the mentioned extended class. Afterwards, we can easily add much more contents to the game like new map or new game mode etc.

4.2.1.2 Singleton Design Pattern

Singleton Design Pattern is likely to be used when any class is supposed to have only one instance. Using such reasoning, we have some classes which behaves like mentioned but we implemented them in static class structure instead of singleton, that is, all attributes and all methods are static, in which it is meaningless to create instances. For this manner, it is like that we have one instance for such class in the whole program domain. The reason for why we used static class instead of singleton is basically much more easier to access and use instance among the program. The statically constructed classes in our design are Game, Turn, GameController and MainApplication since only one instance of those should be involved among the application. Also, we have statically constructed AnimationHandler class, which keeps all the animation instances of the program and performs its functionality through static methods. !!!We need such class because we had to have an handler/manager class to control all animations and defines the relations of animation instances with each other or outer classes.!!! With respect of memory efficiency, we implemented initialize and destroy methods to allocate and deallocate the memory.

4.2.2 Structural Design Patterns

4.2.2.1 Facade Design Pattern

Facade Design Pattern is known as a design pattern which helps designers to hide complexity of the systems and provides a huge but a single interface that users can interact with. It increases the simplicity of the program by making that class to handle all kinds of inputs and outputs in that particular subsystem. For such reasoning, we can state that our Game class is constructed like this because all model instances at the bottom of game should be manipulated in a specific scenario regarding with the input. That is, the game class performs the manipulation of instances regarding with incoming signals from the UI components. For instance, if there is incoming request for action, then Game class firstly checks in which turn phase the game is. Let's say it's attack phase and then game class performs the attack scenario over player, territory and combat instances. For such manner, Game class behaves like a facade, which prevents the UI components directly control the model data, instead, provides them the execution of scenarios over model data.

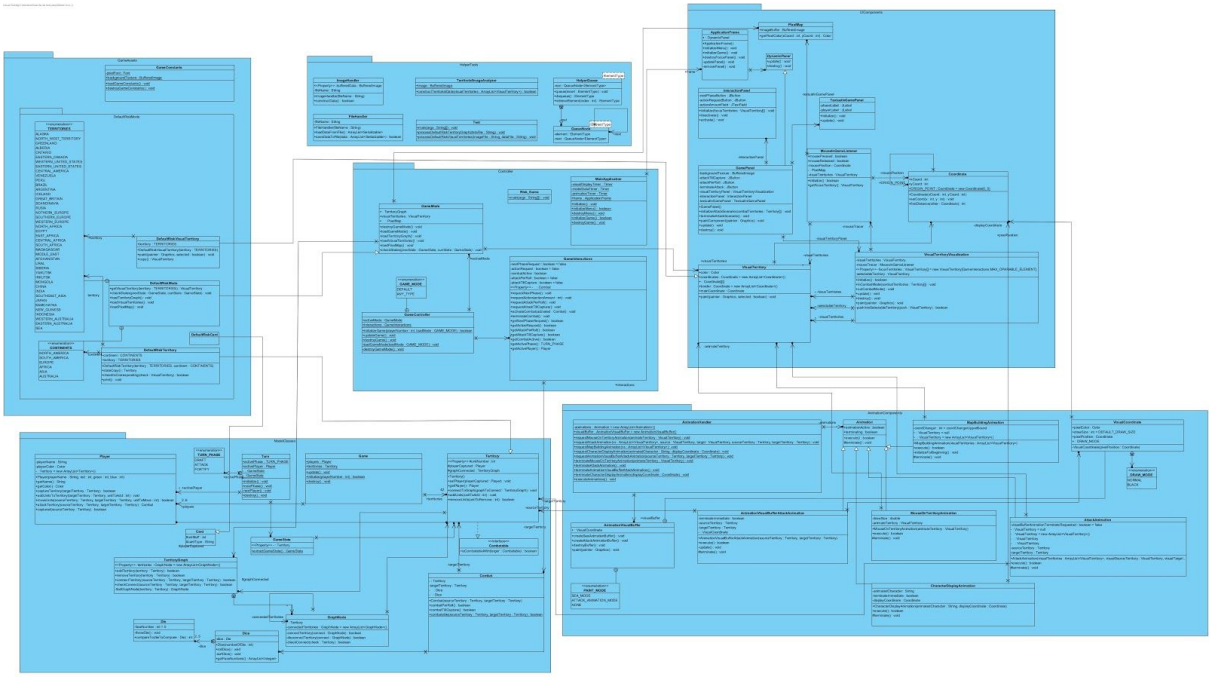
4.2.3 Behavioral Design Pattern

4.2.3.1. Observer Design Pattern

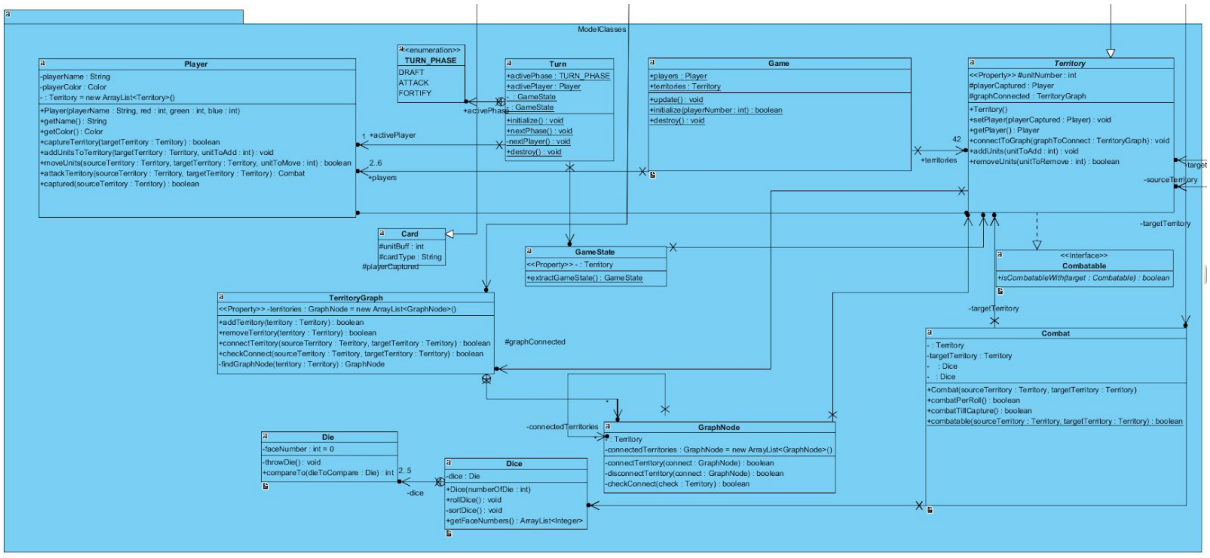
In Observer Design Pattern, any change on the state of subject instance is reflected to observer instances. In this respect, any state change on GamePanel instance is reflected to the GameController class' GameInteractions instance constantly. Similarly, any change on this instance is reflected to the Game class. For this manner, GamePanel is the subject and regarding with incoming inputs, it always notifies the Game class to perform incoming requests from user. That is, we may state that our Game class is an observer class, which is observing the change on UI requests and states.

4.3 Packages

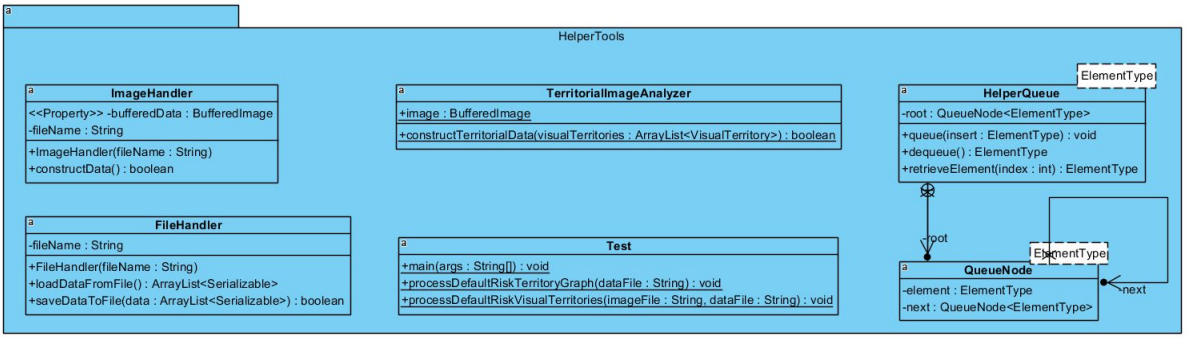
4.3.1. Packages Introduced by Developers



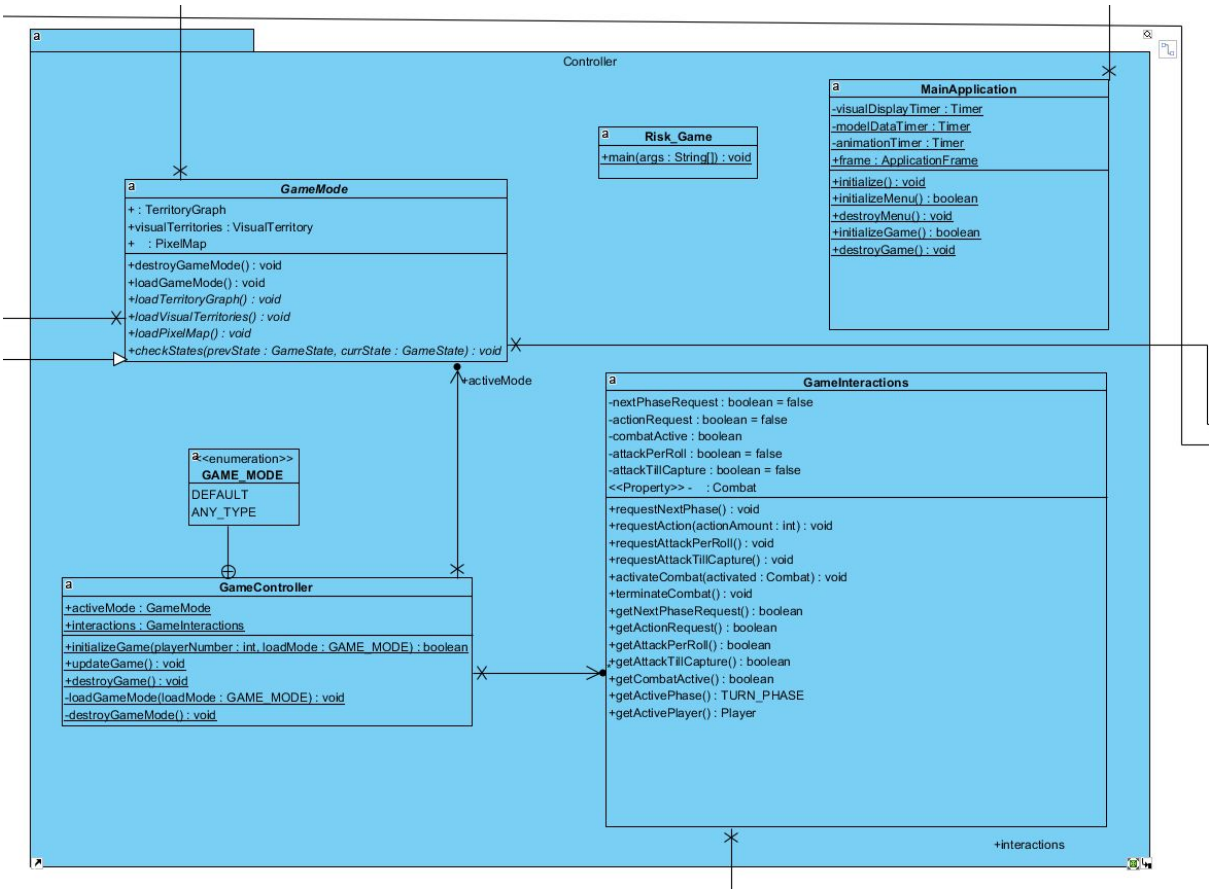
Model Classes



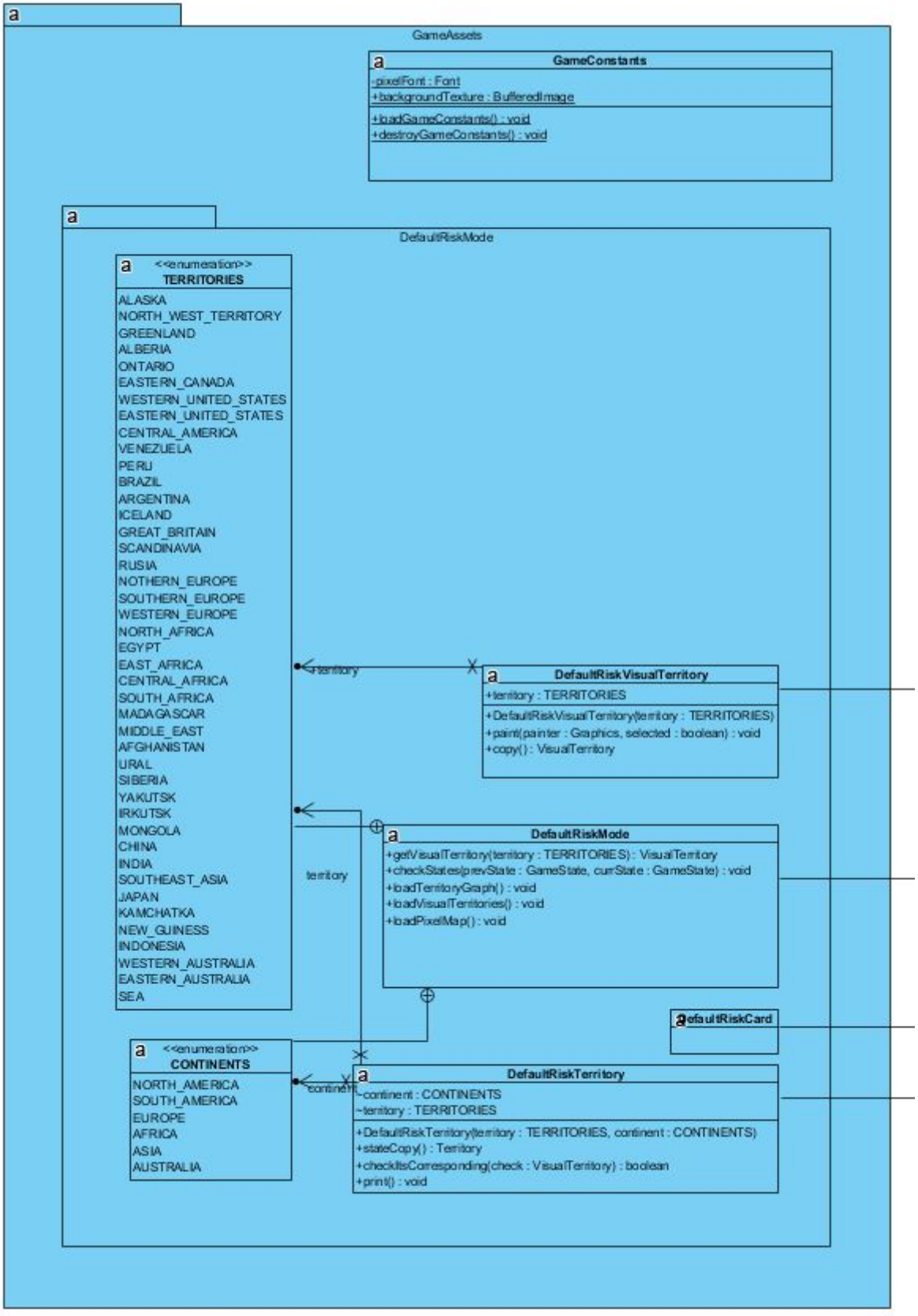
HelperTools



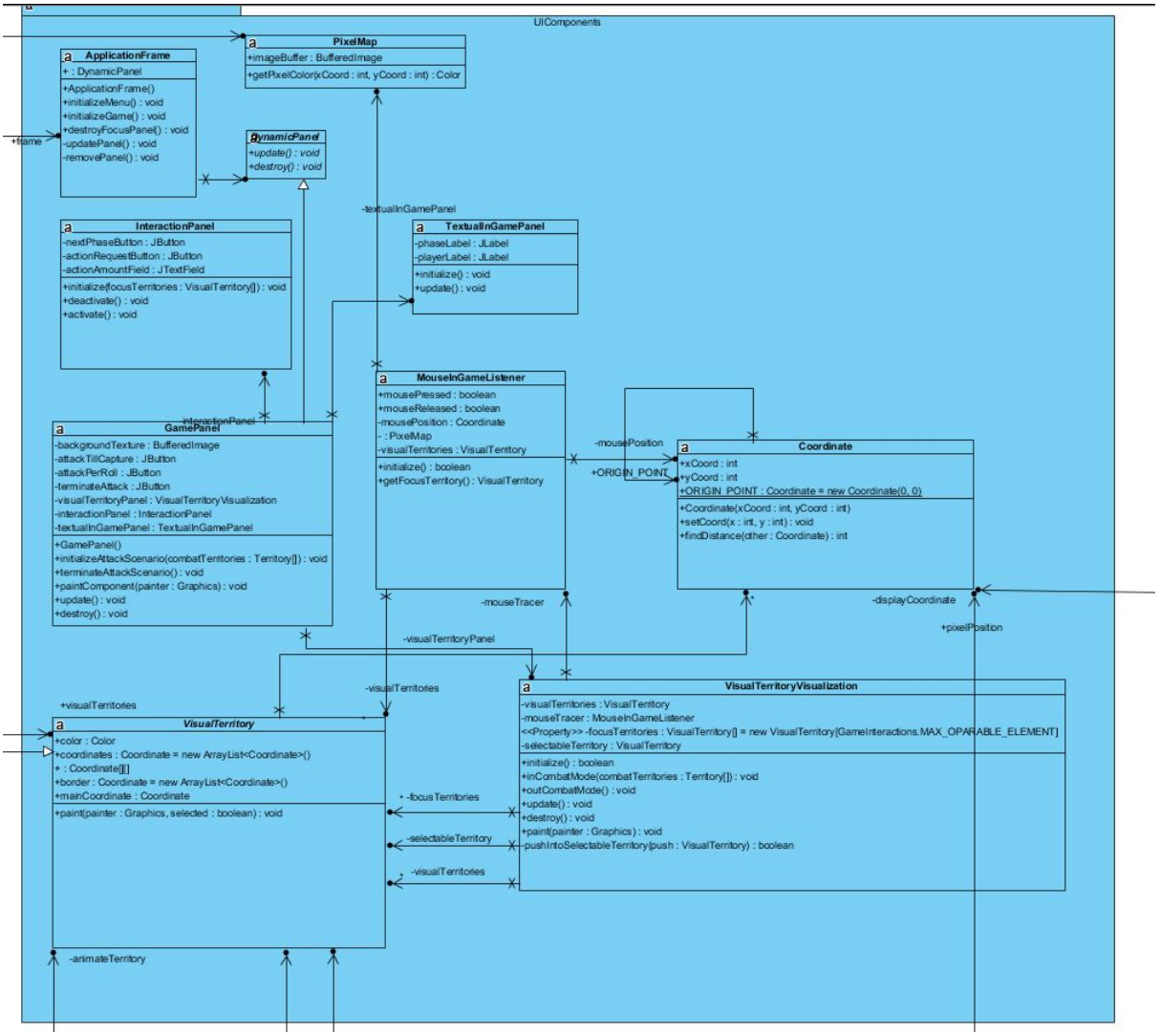
Controller



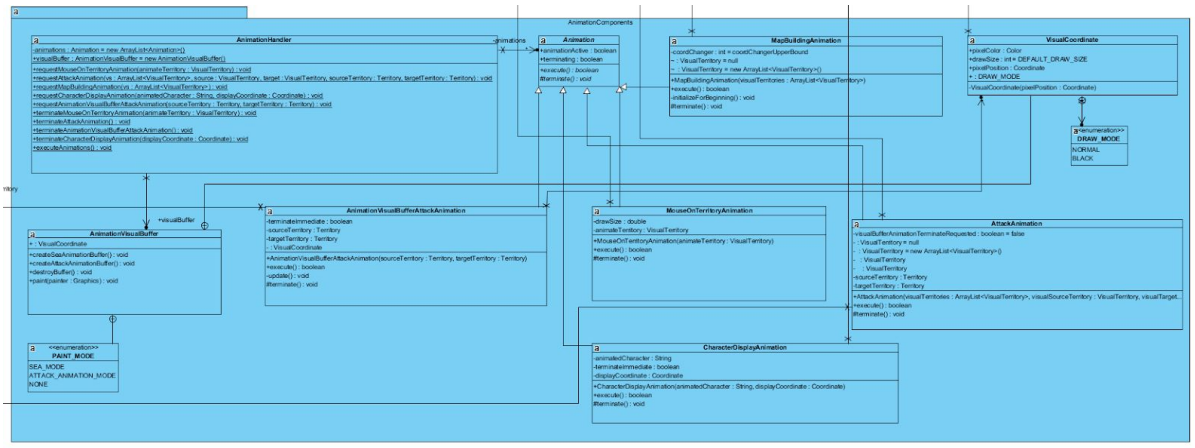
GameAssets


GameAssets
GameConstants
-pixelFont : Font
+backgroundTexture : BufferedImage
+loadGameConstants() : void
+destroyGameConstants() : void
DefaultRiskMode
<<enumeration>>
TERRITORIES
ALASKA
NORTH_WEST_TERRITORY
GREENLAND
ALBERIA
ONTARIO
EASTERN_CANADA
WESTERN_UNITED_STATES
EASTERN_UNITED_STATES
CENTRAL_AMERICA
VENEZUELA
PERU
BRAZIL
ARGENTINA
ICELAND
GREAT_BRITAIN
SCANDINAVIA
RUSIA
NOTHERN_EUROPE
SOUTHERN_EUROPE
WESTERN_EUROPE
NORTH_AFRICA
EGYPT
EAST_AFRICA
CENTRAL_AFRICA
SOUTH_AFRICA
MADAGASCAR
MIDDLE_EAST
AFGHANISTAN
URAL
SIBERIA
YAKUTSK
IRKUTSK
MONGOLA
CHINA
INDIA
SOUTHEAST_ASIA
JAPAN
KAMCHATKA
NEW_GUINESS
INDONESIA
WESTERN_AUSTRALIA
EASTERN_AUSTRALIA
SEA
territory
DefaultRiskVisualTerritory
+territory : TERRITORIES
+DefaultRiskVisualTerritory(territory : TERRITORIES)
+paint(painter : Graphics, selected : boolean) : void
+copy() : VisualTerritory
territory
DefaultRiskMode
+getVisualTerritory(territory : TERRITORIES) : VisualTerritory
+checkStates(prevState : GameState, currState : GameState) : void
+loadTerritoryGraph() : void
+loadVisualTerritories() : void
+loadPixelMap() : void
DefaultRiskCard
<<enumeration>>
CONTINENTS
NORTH_AMERICA
SOUTH_AMERICA
EUROPE
AFRICA
ASIA
AUSTRALIA
continent
DefaultRiskTerritory
-continent : CONTINENTS
-territory : TERRITORIES
+DefaultRiskTerritory(territory : TERRITORIES, continent : CONTINENTS)
+stateCopy() : Territory
+checkItsCorresponding(check : VisualTerritory) : boolean
+print() : void

UIComponents



AnimationComponents



4.4 Class Interfaces

4.4.1. GameController Class

Controller class for manipulating game-model data. Its most essential functionalities are initializing the instances of model classes and updating these instances among the program.

Attributes**:**

- **public static GameMode activeMode:** Stores the GameMode currently active.

- **public static GameInteractions interactions:** Stores the instance of GameInteractions class to enable Game and GamePanel to access such instance.

Methods**:**

- **Public static boolean initializeGame(playerNumber: int, loadMode: GAME_MODE):** Game_Mode is an enum to define possible GameMode. Such methods initializes statically constructed Game class and initializes activeMode attribute by an instance of GameMode regarding with specified GAME_MODE.

- **Public static void updateGame():** Basically calls the update functions of Game and GamePanel classes to update the data of game.

- **Public static void destroyGame():** Deallocates the memory from the initialized data; it is required since GameController statically constructed class.

- **Private static void loadGameMode(loadMode: GAME_MODE):** As understood from being private, its helper method to implement initializeGame(). Just creates and assign an instance to the activeMode with specified mode.

- **Private static void destroyGameMode():** This one helper of destroyGame() method. Deallocates the initialized activeMode by calling its destroy method.

4.4.2. GameInteractions Class

GameInteractions class refers to signal transfer from UI components to the model Game class and sometimes reversely for data matching. Basically it has request method to activate corresponding signal and it will reflects activated signal's effect to the Game class through called getter methods. Signals may be thought as instant data transferring but GameInteractions also keeps the active Combat instance if there exist.

Attributes**:**

- **Private boolean nextPhaseRequest:** Whether next phase is requested or not.

- **Private boolean actionRequest:** Whether an action is requested or not. Action is generalization of Add, Attack and Move actions if action requested Game class performs one of them by checking Turn class's active phase.

- **Private boolean combatActive:** Whether an attack scenario, that is, a combat is active. Game class check this for whether an attack scenario initialized.

- **Private boolean attackPerRoll:** It is attack specific signal corresponds to whether attack per roll is requested or not.

- **Private boolean attackTillcapture:** Same with attackPerRoll but corresponds to attack until capture.

- **Private Combat activeCombat:** Stores the active Combat instance if there exist. Laterly, Game class access it and performs required action on it.

- **private int actionAmount:** Represents the requested action amount. Later on, Game class may extract it to perform action properly.

Methods**:**

- **Public void requestNextPhase():** Game Panel class calls this to request a next phase. Basically makes true corresponding signal.

- **Public void requestAction(actionAmount: int):** Game Panel class calls this to request a next phase. Basically makes true corresponding signal; action amount transferred to local attribute.

- **Public void requestAttackPerRoll():** Game Panel class calls this to request a attack per roll. Basically makes true corresponding signal.

- **Public void requestAttackTillCapture():** Game Panel class calls this to request a attack until capture. Basically makes true corresponding signal.

- **Public void activateCombat(activated: Combat):** Game class calls this since Combat instance generated at the model subsystem. Initializes an attack scenario with specified Combat instance, which is transferred to local attribute; makes true combatActive attribute. Also such method calls GamePanel' initializeAttackScenario() to reflect visual effects.

- **Public void terminateCombat():** Game class calls this if an combat resulted or Game Panel class calls if terminate attack requested. It terminates the attack scenario by deallocating the Combat instance and making combatActive false. Also calls GamePanel' terminateAttackScenario() to reflect visual effects.

- **Public boolean getNextPhaseRequest():** Game class calls this to check next phase is requested. If so, that is, true make it false again and returns true to Game class.

- **Public boolean getActionRequest():** Game class calls this to check action is requested. If so, that is, true make it false again and returns true to Game class.

- **Public boolean getAttackPerRoll():** Game class calls this to check attack per roll is requested. If so, that is, true make it false again and returns true to Game class.

- **Public boolean getAttackTillCapture():** Game class calls this to check attack until capture is requested. If so, that is, true make it false again and returns true to Game class.

- **Public boolean getCombatActive():** Game class calls this to check whether the game is in attack scenario, which a combat is active currently.

- **Public Combat getActiveCombat():** Game class calls this to get active Combat instance to perform required action on it.

- **Public TURN_PHASE getActivePhase():** Game Panel class call this to get which turn phase the game is. Actually it gets such data from Turn class but it is purposed as middle-layer method to follow MVC design.

- **Public Player getActivePlayer():** Game Panel class call this to get which player has the turn. Actually it gets such data from Turn class but it is purposed as middle-layer method to follow MVC design.

4.4.3. GameMode Class

As an abstract it defines which attribute an game mode should have. In this sense, it should keep TerritoryGraph instance, VisualTerritory list and PixelMap instance. It has

methods for initializing and loading those instances, reversely destory methods for deallocating purpose.

Attributes**:**

- **Public TerritoryGraph territoryGraph:** It defines which territories are involved in such game mode and how they are connected with each other to perform game mechanics over such territories.

- **Public ArrayList<VisualTerritory> visualTerritories:** It stores the visual data of territories in territory graph as VisualTerritory. Later on GamePanel extract these data by accessing this.

- **public PixelMap pixelMap:** PixelMap instance to Mouselistener extract it. See PixelMap class.

Methods**:**

- **public void destroyGameMode():** Deallocates the initialized data of local attributes.

- **public void loadGameMode():** Allocates the memory for local attributes initialization. Calls the load methods mentioned below to initialize such variables.

- **public abstract void loadTerritoryGraph():** As an abstract method, it is with purpose of defining how to load territoryGraph attribute. Each extended class of this class may have different file name to load attributes, that is, different implementation.

- **public abstract void loadVisualTerritories():** Similarly loadTerritoryGraph(), as an abstract method, it is with purpose of defining how to load visualTerritories attribute.

- **public abstract void loadPixelMap():** Similarly loadTerritoryGraph(), as an abstract method, it is with purpose of defining how to load pixelMap attribute.

- **public abstract void checkStates(prevState: GameState, currState: GameState):** Such abstract methods is called to perform card mechanics in the game. Since every extended game Mode have different card mechanics, its implementation left to the child class.

4.4.4. MainApplication Class

It is most top level class, that is, it defines how the application goes into different states like menu, game states etc. It keeps the timers to send signals for render, update and animation executions. Also, ApplicationFrame instance to keep visual subsystem but no need for model subsystem since Game class is statically constructed.

Attributes**:**

- **Private static Timer visualDisplayTimer:** It sends signals to ApplicationFrame to render next frame.

- **Private static Timer modelDataTimer:** It sends signals to update model data in the game.

- **Private static Timer animationTimer:** It sends signals to execute animations. See Animations class to understand animation mechanisms.

- **Public static AplicationFrame frame:** The displayed frame of application.

Methods**:**

- **Public static void initialize():** Initializes the timers and frame attributes. It is required since being statically constructed class.

- **Public static boolean initializeMenu():** Initializes menu state of application; if in state of game calls destroyGame().

- **Public static void destroyMenu():** Calls the required destroy() methods of some classes for deallocation of memory from menu-related data**.**

- **Public static boolean initializeGame():** Initializes the game data through calling GameController initializeGame() and ApplicationFrame initializeGame(). After initialization, it restarts the timer of displayTimer and modelDataTimer.

- **Public static void destroyGame():** Calls the required destroy() methods of some classes for deallocation of memory from game-related data.

4.4.5. RiskGame Class

Methods**:**

- **Public static void main(String[] args):** Main method, where the application started the execution. It first calls MainApplication initialize() method and then initializeMenu() method.

4.4.6. DefaultRiskCard Class

Attributes**:**

- **Public DefaultRiskTerritory.TERRITORIES territory:** shows which territory the card corresponds.

- **Public DefaultRiskTerritory.CONTINENTS continent:** shows which continent the card corresponds.

Constructors**:**

- **Public DefaultRiskCard(territory : DefaultRiskTerritory.TERRITORIES, cardType : CARD_TYPE):** Such constructor used when a territory card initialized.

- **Public DefaultRiskCard(continent : DefaultRiskTerritory.CONTINENTS, cardType : CARD_TYPE):** Such constructor used when a continent card initialized.

4.4.7. DefaultRiskMode Class

It stores the name of files where the mode-specific data stored. That is, the related methods load them by using such file names. It also stores the enums called TERRITORIES and CONTINENTS to specify which territory and continents involved in DefaultRiskMode.

Methods**:**

- **Public void checkStates(prevState: GameState, currState: GameState):** Executes the card mechanics by comparing two state and checking the buff amount of related Card instances.

- **Public void loadTerritoryGraph():** From the locally stored file, loads the TerritoryGraph instance, that is, also territories list and their connections in a sense.

- **Public void loadVisualTerritories():** From the locally stored file, loads the VisualTerritory instances, that is, the visual data of territories.

- **Public void loadPixelMap():** From the locally stored file, loads the Pixel Map instance, that is, also territories list and their connections in a sense.

4.4.8. DefaultRiskTerritory Class

Attributes**:**

- **Public DefaultRiskMode.CONTINENTS continent:** Shows which default risk continent it belongs to.
- **Public DefaultRiskMode.TERRITORIES territory:** Shows which default risk territory it belongs to.

Constructors**:**

- **Public DefaultRiskTerritory(territory: TERRITORIES, continent: CONTINENTS):** Creates an instance of default risk territory with specified territory and continent.

Methods**:**

- **Public Territory stateCopy():** Creates a copy instance which holds the same amount of units and player.
- **Public boolean checkItsCorresponding(check: VisualTerritory):** Returns whether specified visual territory corresponds to such territory instance basically it checks their territory attribute is same or not.
- **Public void print():** Prints the properties of territories.

4.4.9. DefaultRiskVisualTerritory Class

Attributes**:**

- **public DefaultRiskMode.TERRITORIES territory:** Shows which default risk territory it belongs to.

Constructors**:**

- **public DefaultRiskVisualTerritory(territory: TERRITORIES):** Construct default risk visual territory instance with specified territory.

Methods**:**

- **Public void paint(painter: Graphics, selected: boolean):** It calls parent class paint methd and also checks forsome properties.
- **Public VisualTerritory copy():** Creates copy instance which holds same coordinates with its own list that is a copy visual data.

4.4.10. Test Class

Methods**:**

- **Public static void main(args: String[]):** While developing the game some time side programs are needed, Test class executes such programs.

4.4.11. Card Class

Attributes**:**

- **Public CARD_TYPE cardType:**Type of the card

Constructors**:**

- **Public Card(cardType : CARD_TYPE):** Which types are in the game

Methods**:**

- **Public getBuffAmount():**How much buff thet card gives to the player

4.4.12. Combat Class

Attributes**:**

- **Private Territory souceTerritory:**The attacking territory in the combat
- **Private Territory targetTerritory:**The defending territory in the combat
- **Private Dice attackDice:**The attacking territories dice
- **Private Dice defendDice:**The defending territories dice

Constructors**:**

- **Public Combat (sourceTerritory: Territory, targetTerritory: Territory):**Creates a combat

Methods**:**

- **Public boolean combatPerRoll():**Executes the combat for just once and deletes the dead units.Incase the territory is captured moves the units to the captured territory
- **Public boolean combatTillCapture():**Executes the combat for one of the territory is captured , deletes the dead units and moves the units to the captured territory

- **Public boolean combatable(sourceTerritory: Territory, targetTerritory: Territory):**It checks whether two chosen territories can attack each other by looking their position in the graph and the owner of the territories.

4.4.13. Combatable Interface

Methods**:**

- **Public boolean isCobatableWith(target: Combatable):**Checks does targetTerritory is combatable

4.4.14. Dice Class

Attributes**:**

- **Private Die dice:**Dice

Constructors**:**

- **Public Dice(numberOfDie: int):**

Methods**:**

- **Public void rollDice():** this method generates a random variable
- **Private void sortDice():**sorts the dices in order
- **Public ArrayList<Integer> getFaceNumbers():** Output of the dices

4.4.15. Game Class

Attributes**:**

- **Public static Player players:**Player in the current game
- **Public static Territory territories:**Territories in the current Game

Methods**:**

- **Public static void update():**Updates the game 100 times in a second
- **Public static boolean initialize(playerNumber: int):**Initializes the game in the opening
- **Public static void destroy():**Closes the game

4.4.16. GameState Class

Attributes**:**

- **private ArrayList<Territory> territoriesState:**Current state

Methods**:**

- **Public static GameState extractGameState()**Changes in the state of the game

4.4.17. Player Class

Attributes**:**

- **Private String playerName:** tname of the player
- **Private Color playerColor:** the color that is assigned to the player
- **Private Territory territories:** Territories that player have

Constructors**:**

- **Public Player(playerName: String, red: int, green: int, blue: int):**

Methods**:**

- **Public String getName():** this method returns the name of the player
- **Public Color getColor():** this method return the color that is assigned to the player
- **Public boolean captureTerritory(targetTerritory: Territory):** this method adds a specific territory to a player and shows it on map
- **Public void addUnitsToTerritory(targetTerritory: Territory, unitToAdd: int):** this method adds units to the specific territory of the player
- **Public boolean moveUnits(sourceTerritory: Territory, targetTerritory: Territory, unitToMove: int):** this method moves units from one territory of the player to another
- **Public Combat attackTerritory(sourceTerritory: Territory, targetTerritory: Territory):** this method is used when It is possible player attacks to the other territories
- **Public boolean captured(sourceTerritory: Territory):** this method detects if player captured the attacked territory or not

4.4.18. Territory Class

Attributes**:**

- **Protected Player playerCapture:** Player who has the player
- **Protected TerritoryGraph graphConnected:** Territories that are connected

Constructors**:**

- **Public Territory():**

Methods**:**

- **Public void setPlayer(playerCaptured: Player):**This method sets the player to a specific territory
- **Public Player getPlayer():**this method returns the player who owns the specific territory
- **Public void connectToGraph(graphToConnect: TerritoryGraph):**this method connects two specified territories with each other, which makes them boundary related territories, that is, the interaction over such territories is possible afterwards

- **Public void addUnits(unitToAdd: int):** this method is used to add units to a specific territory
- **Public boolean removeUnits(unitToRemove: int):** this method removes units from a specific territory

4.4.19. TerritoryGraph Class

Attributes**:**

- **Private Arraylist<GraphNode> territories:**

Methods**:**

- **Public boolean addTerritory(territory: Territory):** this method adds a territory into specified territory graph
- **Public boolean removeTerritory(territory: Territory):** Remove territory from the graph
- **Public boolean connectTerritory(sourceTerritory: Territory, targetTerritory: Territory):**this method connects two specified territories with each other, which makes them boundary related territories, that is, the interaction over such territories is possible afterwards
- **Public boolean checkConnect(sourceTerritory: Territory, targetTerritory: Territory):** this method returns whether two territories connected in terms of boundary relation to use such data further operations in the game
- **Private GraphNode findGraphNode(territory: Territory):** this method finds the node in graph

4.4.20. Turn Class

Attributes**:**

- **Public static TURN_PHASE activePhase:** Turn holld the current state of the game.
- **Public static Player activePlayer:**which player is active at the time
- **Private static GameState currentState:**In order to handle the changes in game compere the current state with the previous state
- **Private static GameState prevState:**Previous state

Methods**:**

- **Public static void initialize():**which player is active at the time Initializes the first turn
- **Public static void nextPhase():**Changes the phase attack,Moveunit,Addunit.
- **Private static void nextPlayer():**Changes the active player
- **Public static void destroy():**Destroy the turn while closing the game

4.4.21. AboutUsPanel Class

Attributes**:**

- **private JLabel lblBack:**When user goes to the **How To Play** he needs to go back this label is used to go back from where he came. This label has listeners which does some action.

- **private ArrayList<DrawingString> stringList:** The list keeps the strings that will be drawn to the panel.

- **private Fireworks fireWorks:** This is a fireworks animation instance.

Constructors**:**
- **public AboutUsPanel():** Initializes the string list.

Methods**:**
- **public void initialize():** Adds the strings to the list which will be drawn.
- **public void paintComponent(Graphics g):** Draws the strings.

4.4.22. ApplicationFrame Class
Attributes**:**
- **Public DynamicPanel focusPanel:** In the frame there will be only one panel which will be active. This instance holds that panel.

Constructors**:**
- **Public void ApplicationFrame():** Is used when the frame object is created, does not do any specific operations.

Methods**:**
- **Public void initializeMenu():** When this method is called the Menu panel is created and focused.
- **Public void initializeGame():** When this method is called the game panel is created and focused.
- **Public void destroyFocusPanel():** Destroys the panel which is focused at the moment.
- **Private void updatePanel():** Updates the focused panel.
- **Private void removePanel():** Removes the selected panel from the frame.

4.4.23. ApplicationPanel Class
Except than game panel all other panels are controlled via this class. By using the listeners.

Attributes**:**
- **private JPanel self:** Shows the current panel.
- **private JPanel menuPanel:** An instance of Menu Panel.
- **private JPanel settingsPanel:** An instance of Settings Panel.
- **private JPanel howToPlayPanel:** An instance of How To Play Panel.
- **private JPanel aboutUsPanel:** An instance of About Us Panel.

4.4.24. Coordinate Class
Attributes**:**
- **Public int xCoord :** represents the x coordinate in 2D.
- **Public int yCoord :** represents the y coordinate in 2D.

- **Public static Coordinate ORIGIN_POINT :** represents the origin(0,0) in 2D

Constructors**:**

- **Public Coordinate(xCoord: int, yCoord: int) :** Creates an instance of Coordinate class with specified parameters

Methods**:**

- **Public void setCoord(x: int, y:int) :** Sets the coordinates xCoord and yCoord to x and y.
- **Public int findDistance(other: Coordinate):** Finds the distance between the original coordinate and the given coordinate.

4.4.25. DynamicPanel Class

This class extends JPanel.

Methods**:**

- **public abstract void update():** Some classes, which are panels as well, that extends this class uses update() method to update data in the current panel that is executed.
- **public abstract void destroy():** During run time when we switch in between panels in order not to waste memory we destroy the previous panel that was running.

4.4.26. GamePanel Class

Attributes**:**

- **Private JButton attackTillCapture:** During the attack phase if player wants to attack until he captures another territory this button is pressed.
- **Private JButton attackPerRoll:** Unlike attackTillcapture button user can press once to this button and either he captures the territory he was attacking or not according to the results he can terminate the attack.
- **Private JButton terminateAttack:** User can cancel the attack process during fight by using this button.
- **Private VisualTerritoryVisualization visualTerritoryPanel:** This attribute contains a visual territory, draws the territory and holds the mouse listener that we use in game.
- **Private InteractionPanel interactionPanel:** In game we have some buttons that is used during the game play for the interactions of attack, fortify and add phases.

Constructors**:**

- **Public GamePanel():** Constructor initializes the in game panel, with buttons phases names and map.

Methods**:**

- **Public void initializeAttackScenario(combatTerritories: Territory[]):** If user chooses to attack we hide other buttons and disable them. At the same time the UI changes a little bit. It is done by this method.
- **Public void terminateAttackScenario():** As mentioned before user can terminate the fight if he wants. When user terminates the attack UI changes back to the in game mode. Which is done by this method.

- **Public void paintComponent(painter: Graphics):** We call the **paint** method of visualTerritoryPanel here in order to draw and updates the map, renders the map in one word.
- **Public void update():** Updates the texts in the panel, for example when the **turn** changes name of the players changes and phases also change according to that.
- **Public void destroy():** Destroys the panel.

4.4.27. HowToPlayPanel Class

Attributes**:**

- **private JLabel lblHowToPlay:** How to play text on the panel.
- **private JLabel lblBack:** When user goes to the **How To Play** he needs to go back this label is used to go back from where he came. This label has listeners which does some action.
- **private ArrayList<DrawingString> stringList:** In order to keep the overall game theme single, we write the info on how to play the game by drawing strings. This list keeps those information.
- **private Fireworks fireWorks:** This is a fireworks animation instance.

Constructors**:**

- **Public HowToPlayPanel():** Initializes the **back** label, and creates the string list.

Methods**:**

- **Public void initialize():** When this method is called all the info about how to play is inserted in to the string list where we keep the information.
- **Public void paintComponent():** When this method is called all the info about how to play is drawn on the panel. In order not to loose animation we are not doing this with constructor.

4.4.28. MenuPanel Class

Attributes**:**

- **private JLabel lblPlayGame:** When this label is pressed game starts.
- **private JLabel lblQuit:** When this label is pressed the application is closed.
- **private JLabel lblSettings:** When this label is pressed user goes to the settings.
- **private JLabel lblHowToPlay:** When this label is pressed user goes to the how to play.
- **private JLabel lblAboutUs:** When this label is pressed user can see information about us.
- **private Pixel mapAnimation:** We have a map animation in menu panel. Which is the instance of Pixel class.
- **private ArrayList<DrawingString> stringList:** Same as previous classes we draw the strings on menuPanel. This list holds those strings.

Constructors**:**

- **Public MenuPanel():** Initializes the string list, and other labels which are used to go from one panel to another.

Methods**:**

- **Public void initialize():** Strings which will be drawn when user goes to the menu panel are added to the list when this method is called.
- **Public void paintComponent():** We draw the strings by this method. Also, we draw the map animation via this method.

4.4.29. MouseInGameListener Class

Attributes**:**

- **Public boolean mousePressed:** This attribute keeps data whether mouse is pressed or not
- **Public boolean mouseReleased:** This attribute keeps data whether mouse is released or not
- **Private Coordinate mousePosition:** This attribute simply keeps the position of mouse on the screen
- **Private VisualTerritory visualTerritories:** The VisualTerritory instance.

Methods**:**

- **Public boolean initialize():** Activates the pixel map and visual territories.
- **Public VisuallTerrirtory getFocusTerritory():** Returns the current territory which is selected at the moment.

4.4.30. OptionsPanel Class

Attributes**:**

- **private JLabel labelSettingsGif:** A smple settings gif, something like image, does not do anything specific.
- **private JComboBox comboBoxMusic:** A combobox used to store the in game sounds and musics.
- **private JComboBox comboBoxTurnOnOffMusic:** User can turn on/off music, via selecting it from the combobox inputs.
- **private JComboBox comboBoxChangeDifficulty:** In the single game mode where player plays with AI, the game will have some difficulty levels, which could be changed via selecting it from the inputs of combobox.
- **private JLabel lblBack:** When pressed this button user goes to back where he came from to the options panel.
- **private ArrayList<DrawingString> stringList:** The list keeps the strings that will be drawn to the panel.
- **private Fireworks fireWorks:** Instance of fireworks animation.

Constructors**:**

- **Public OptionsPanel():** String list is initialized here, as well as the labels and comboboxes.

Methods**:**

- **Public boolean initialize():** The strings that will be drawn is added to the list when this method is called.
- **Public void paintComponent():** We draw the strings by this method.

4.4.31. PixelMap Class

Attributes**:**

- **Public BufferedImage imageBuffer:** It holds the color of the each coordinate in 2D array.

Methods**:**

- **Public getPixelColor(xCoord : int, yCoord : int):** Returns the color of the given coordinate.

4.4.32. VisualTerritory Class

Attributes**:**

- **Public Color color:** This attribute keeps the color of visualized territory
- **Public ArrayList<Coordinate> coordinates:** This arraylist keeps the set of coordinates that belong to the visualized territory
- **Public ArrayList<Coordinate> border:** This arraylist keeps the border coordinates of the visualized territory

Methods**:**

- **Public void paint(painter: Graphics, selected : boolean):** Paints the territories.

4.4.33. VisualTerritoryVisualization Class

This class hold the data of the territories in or the show it in the screen

Attributes**:**

- **Private VisualTerritory visualTerritories:**Visual data of the territories
- **Private MouseInGameListener mouseTracer:**In order to execute a animation gets the mouse position
- **Private VisualTerritory[] focusTerritories:**Selected Territory
- **Private VisualTerritory selectableTerritory:**Selectable territory impies the territories you can attac k and move unit

Methods**:**

- **Public boolean initialize():**Initially load the data of the territories and puts it in the program
- **Public void inCombatMode(combatTerritories : Terrirory[]):**This method is here to execute show combat animation
- **Public void outCombatMode():**This method is here to endshow combat animation
- **Public void update():**Updates the territory visualization changes
- **Public void destroy():**Destroys the object while closing the game
- **Public void paint(paintr: Graphics):**Paints the All therritories
- **Public boolean pushIntoSelectableTerritory(push : VisualTerritory):**Used for showing the selectable territories

4.4.34. Animation Class

Attributes**:**

- **Public boolean animationActive:**Hold b active animation
- **Public boolean terminating:**Is terminating the active animation is still active

Methods**:**

- **Public boolean execute():**Starts the execution of the animation
- **Protected void terminate():**Terminates the execution of the animation

4.4.35. AttackAnimation Class

Attributes**:**

- **Private VisualTerritory target:** This attribute keeps the visual of target territory in attack animation
- **Private VisualTerritory source:** This attribute keeps the visual of attacking territory in attack animation
- **Private ArrayList<VisualTerritory> visualTerritories:Private Territory sourceTerritory:** This attribute keeps the attacking territory in attack animation
- **Private Territory targetTerritory:** This attribute keeps the target territory in attack animation

Constructors**:**

- **AttackAnimation(visualterritories: ArrayList<VisualTerritory>, visualSourceTerritory: VisualTerritory, visualTargetTerritory: VisualTerritory, sourceTerritory: Territory, targetterritory: Territory):** This constructor simply initializes all the attributes by given input objects

Methods**:**

- **Public boolean execute():** This method executes the animation
- **Protected void terminate():** This method terminates the animation

4.4.36. MapBuildingAnimation Class

Attributes**:**

- **private int coordChanger:** Animation modifier which determines the speed of the animation.
- **public ArrayList<VisualTerritory> visualTerritories:** To manipulate the data of the territory in the list.

Constructors**:**

- **Public MapBuildingAnimation(visualTerritories: ArrayList<VisualTerritory>):** Copies the input visualTerritories to the class attribute visualTerritories.

Methods**:**

- **Public boolean execute():** Executes its charasteristic animation over manipulated territories. It should be considered as only one wave of animation execution.
- **Protected void terminate():** To reverse the effect of the animation by returning the data of manipulated instance to the original state.

4.4.37. MouseOnTerritoryAnimation Class

Attributes**:**

- **Private double drawSize:** Size of the selection
- **Private VisualTerrittory animateTerritory:** While the mouse is on the territory animateTerritory hold the which territory it is

Constructors**:**

- **Public MouseOnTerritoryAnimation(animateTerritory: VisualTerritory):Animation** Constructor

Methods**:**

- **Public boolean execute():**Executes the animation
- **Protected void terminate():**Terminates the animation

4.4.38. DrawingString Class

Attributes**:**

- **public Boolean end:** To determine whether the animations are done.
- **public int x, y, minLength, maxLength:** Top left coordinates and border length of squares
- **public String string:** Written string.
- **public ArrayList<Rect> list:** Contains all squares
- **public int startAnimationDelay, startAnimationSpeed:** Animation speed and delay of squares to move them to their original position.
- **private int endAnimationDelay, endAnimationSpeed:** Animation speed and delay of squares to move them to outside of the screen.
- **public Timer startAnimation, endAnimation, mexicoWave:** Timers of start, end and mexicoWave animations. Coordinates and lengths are changed here.
- **Public int mexicoWaveCoordinate, mexicoWaveDelay, mexicoWaveLength:** Coordinate determines to which coordinate we enlarge the length of squares. Delay is the delay of timer used for this animation. Length is set the difference of max and min length to change them smoothly.
- **Public boolean mouseEntered:** To determine whether the mouse is entered to the string in screen.
- **Public class Rect:** To get the rect object for the mouse entered there.

Constructors**:**

- **public DrawingString(x: int, y: int, maxLength: int, string: String):** Basically draws a string into our screen by using squares. One letter is build from 3 square along x axis and 5 square along y axis. X and y are top left coordinate, maxLength is max length of one square.

Methods**:**

- **public void paint(g: Grapihcs):** Draws squares into our screen.
- **public void makeAreaB(topLeftCoordinate: Rect, length: int):** Adds a new rect into our list in terms of top left coordinate and length.
- **public void makeAreaS(topLeftCoordinate: Rect, length: int):** Does nothing, written just because decreasing the mind blowing caused by drawNumberOn method.
- **public void drawNumberOn(topLeft: Rect, blockLength: int, ch: char):** Adds new rects into our list in terms of specific character. Works like seven segment display. One character is impemented by 3x5.
- **public Rectangle getRectangle():** Gives rectangle to determine mouse in this string in the screen or not.

4.4.39. ExplodeOne Class

Attributes**:**

- **public int borderLength:** Length of the borders of one particule of fireworks after explode.
- **public int xMovement, yMovement, xCenter, yCenter:** xCneter and yCenter is used to determine the center of particule. xMovement and yMovement are alteration of x and y coordinates of a particule throughout animation.
- **public int yMovementChanges:** Decreasing of the speed of particules in y axis caused by gravity.
- **public Color c:** Color of the particule.
- **public int opacity:** Transparency of one particule.
- **public int opacityChanges:** Used to decrease the transparency of one particule.
- **public boolean done:** To determine the delete the object.
- **public int rect:** To determine whether the particule will be rectangle or oval.

Constructors**:**

- **public ExplodeOne(xCenter: int, yCenter: int, xMovement: int, yMovement: int, borderLength: int, c: Color):** Simply creates the particule after a firework explode.

Methods**:**

- **public void paint(g: Graphics):** Draws particule.
- **public void update():** Changes the x and y coordinates, length and transparency of particule.

4.4.40. Fireworks Class

Attributes**:**

- **private int minSpeed, maxSpeed:** Min and max speed of firework. Speed is randomly selected between these values.
- **private ArrayList<RisingOne> risingOneList:** List of fireworks which are to explode.
- **private ArrayList<ExplodeOne> explodeOneList:** List of fireworks which are after explode.

Constructors**:**

- **public Fireworks():** Includes timers to add new fireworks into risingOneList and another timer is used to update the attributes of risingOne and explodeOne objects.

Methods**:**

- **public void paint(g: Graphics):** Draws the objects into our screen.

4.4.41. RisingOne Class

Attributes**:**

- **private int length,yStart xStart, boomYlocation, initialSpeed, currentSpeed:** Length of firework. Initial coordinate of firework as xStart and yStart. boomYlocation is where to explode the firework. InitialSpeed is initial speed of firework. CurrentSpeed is speed in one time.
- **private Color c:** Color of firework.

- **private boolean boom:** To delete the object and add ExplodeOne objects into list in Fireworks class.

Constructors**:**

- **public RisingOne(xStart: int, boomYlocation: int, initialSpeed: int, c: Color):** Creates a firework object which is before explode. It is like single vertical line moving upwards.

Methods**:**

- **public void paint(g: Graphics):** Draws the line.
- **public void update():** To change parameters of object. Used in timer.

4.4.42. Circle Class (For RainMap)

Attributes**:**

- **int xCenter, yCenter, maxRadius, currentRadius, radiusDifference. seaOpacity, seaOpacityDecrement:** Center of circle. Max and current radius of circle in one time. Opacity is transparency of sea wave.
- **boolean done:** To check whether it should be deleted or not.

Constructors**:**

- **public Circle(xCenter: int, yCenter: int):** Adds a circle to determine which coordinates are shown and in the circle.

Methods**:**

- **public void update():** To update the attributes.
- **public boolean isInCircle(c: Coordinate)**: To determine coordinate is in our circle.
- **public void paint(g: Graphics):** paints circle.

4.4.43. Coordinate Class (For RainMap)

Attributes**:**

- **private int y, x:** Center of circle.
- **private boolean move:** To determine movable or not.

Constructors**:**

- **public Coordinate(x: int, y: int):** Creates coordinate.

4.4.44. Pixel Class

Attributes**:**

- **private int red, green, blue, mouseX, mouseY:** rgb color of map. Mouse coordinates.
- **private ArrayList<Coordinate> list = new ArrayList<Coordinate>():** Coordinates for the squares of map.
- **private ArrayList<Coordinate> listCopy = new ArrayList<Coordinate>():** Copy Coordinates for the squares of map.
- **private ArrayList<Coordinate> listSea = new ArrayList<Coordinate>():** Coordinates for the squares excluding map.
- **private ArrayList<Circle> listCircle = new ArrayList<Circle>():** Circles to differentiate the wave which has coordinates including circle.

- **private int x,y, jump:** Jump is length of one square in map or x and y are length of oval in wave.
- **private Color c:** Color of map.
- **public Timer seaMouse:** Used for wave animation.

Constructors**:**

- **public Pixel():** Creates a world map using squares. A random square moving downwards and it is rain effect for map. Also wave effect is running as animation.

Methods**:**

- **public void startDraggedMouse(mouseX: int, mouseY: int):** Creates wave while mouse is dragged.
- **public void addCircle(mouseX: int, mouseY: int):** adds Circle to draw wave. Used in Timer.
- **public void restart():** When map has zero square. Restart map and Retrieve all squares.
- **public void paint(g: Graphics):** Paints everything.
- **public static ArrayList<Coordinate> getCopyList(list: ArrayList<Coordinate>):** get copy of squares used for map.

4.4.45. Circle Class (For sea)

Attributes**:**

- **private int xCenter, yCenter, maxRadius, currentRadius, radiusDifference. seaOpacity, seaOpacityDecrement:** Center of circle. Max and current radius of circle in one time. Opacity is transparency of sea wave.
- **private boolean done:** To check whether it should be deleted or not.

Constructors**:**

- **public Circle(xCenter: int, yCenter: int):** Adds a circle to determine which coordinates are shown and in the circle.

Methods**:**

- **public void update():** To update the attributes.
- **public boolean isInCircle(c: Coordinate)**: To determine coordinate is in our circle.
- **public void paint(g: Graphics):** Paints circle.

4.4.46. Coordinate Class (For sea)

Attributes**:**

- **private int y, x:** Center of circle.
- **private boolean move:** To determine movable or not.

Constructors**:**

- **public Coordinate(x: int, y: int):** Creates coordinate.

5. Glossary

MVC: Model-View-Controller architecture

6. References

https://www.tutorialsteacher.com/mvc/mvc-architecture
https://medium.com/code-smells/model-view-controller-architecture-78a855b8ab56