Design Document for:

Risky Business

Written by
Angela Niu
Craig Williams
Jacob Lehman
Jacob Wortman
Jon Claus
Joseph Pecoraro
Kyle Maharlika
Ricardo Iván Vieitez Parra
Victor Cao

Version # 1.00

1/29/14

1 Introduction

1.1 Goals and objectives

As stated previously in the SRS, the goal of this project is to create an application for tablets with the Android platform, utilizing elements from the popular board games RISK and Settlers of Catan to create a unique game that can be enjoyed by players of many different audiences. The game, titled "Risky Business", will incorporate a hexagonal board layout and resource building system similar to that of Settlers of Catan while also tying in the system of building up a combat force and taking over the opponent's resources, derived from the combat and territorial system found in RISK.

1.2 Statement of scope

1.2.1: Essential functions and aspects

- Ability to poll for and accept input from the player, and determine and call the appropriate action or view
- A functional visual game board with hextiles and textures (managed by the view component), as well as edge,vertex, and player data
- User interface that lists the player's resources and military and can bring up the building and trade screens.
- Function for dice roll, and a resource dispersion system that will automatically distribute resources at the start of every turn after the dice roll.
- Battle system, with interactions between soldiers and other objects of the board.
- Ability to pan and zoom
- Main Menu

1.2.2 Desirable functions and aspects

- Settings Menu (sound and visual quality)
- Help Menu
- A tutorial of sorts, or a demo mode
- Ability to alter the size of the board before starting the game
- Ability to connect to other players and play over WiFi
- Local rankings and statistics

1.2.3 Future requirements

- A functional AI to allow single-player play, as well as varying difficulties for

the AI
- Mini games to enhance gameplay experience

1.3 Software context

The purpose of this software project is to create an Android game for the CEN 3031 class that could possibly be released into the Android app market in the future. This application will only work on tablets due to size constraints that phones cannot fulfill. Due to our current focus developing this game just for the CEN3031 class, competition is not a very large concern for us; if released later into the market, there are some similar games already out in the market, but with our incorporation of elements from RISK, our game is unique enough to stand out.

1.4 Major constraints

1.4.1 Unfamiliarity with Our Development Platform

The members of our team have not worked with Android extensively in previous projects. None of us have extensive experience in GIT – although some of us do have a reasonable amount of experience – and very few of us have delved into the world of developing aesthetics in software. This project, therefore, is somewhat ambitious given the skillset we do have.

1.4.2 Time Constraints

Given that we only have one semester to complete the project at hand, while having to also deal with other work in our lives, the number of features we can reasonably implement will be limited, and as such, some desired but non-critical features may not end up being implemented.

1.4.3 Testing Constraints

In an actual software production environment, this product would have a designated beta test period separate from the development phase, which would include thousands of participants in the test to give feedback about various problems and bugs in the product. We, however, do not have access to such time or resources.

2 Data design

A description of all data structures including internal, global, and temporary data structures.

2.1 Internal software data structure

"Risky Business" shall follow the MVC design pattern, meaning that data structures shall pertain into models. Models shall provide any necessary abstractions for the rest of the software components that interact with them. The following descriptions reflect the design considerations associated with the main game data structures; their implementation, however, may be broken down into several models or multiple structures combined into one.

2.1.1 Board Structure

One of the most important structures within the application is that of the board. The board is the only truly global data structure, and the game engine shall rely heavily on this representation.
Internally, the board shall have:

- a Set of all the game hexagons
- an ArrayList of all players
- a HashMap of the dice rolls-resources mappings
- a HashMap tracking user-resource allocation
- internal, private variables relarted to the particular game, such as the current player, match start time, etc.w

The board shall provide an interface to access the contents of the different structures that it contains. These accesses shall be read-only exclusively, except for those accesses necessary for the game engine to enable game play, which shall be conducted using a writable encapsulation mechanism. Object encapsulation, by means of internal classes or otherwise, shall be enforced to assure that any objects returned are also read-only.

2.1.2 Edge Structure

An edge is a part of an hexagon that joins two vertices together. An edge may be shared between two hexagons, and players may build roads along edges. Edges shall contain these properties as well as an interface to read them:

- an immutable array of Hexagons, having capacity for two slots.
- an immutable array of Vertices, having capacity for two slots.
- a reference to Player, which points to the owner of a road built on the edge. A value of NULL implies that no road has been built on the edge.

2.1.3 Vertex Structure

A vertex is a part of an hexagon that joins up to three edges together. A vertex may be shared among three hexagons, and players may place settlements and station armies on vertices. Vertices shall contain these properties as well as an interface to read them:

- an immutable array of Hexagons, having capacity for three slots.
- an immutable array of Edges, having capacity for three slots.
- a reference to Player, which points to the owner of the elements placed on the vertex. A value of NULL implies that no road has been built on the edge.
- an enumerated type, that describes the type of element placed (e.g., settlement, army, etc.)

2.1.4 Hexagon Structure

A hexagon is the basic board unit. Each hexagon shall have these properties as well as an interface to access them:

- An immutable array of Edges, with capacity for six slots. Edges shall be stored in counterclockwise order, with the first slot occupied by the top-right edge.
- An immutable array of Vertices, with capacity for six slots. Vertices shall be stored in counterclockwise order, with the first slot occupied by the rightmost vertex.
- An immutable array of ajacent Hexagons. The slots shall be occupied in the same fashion as edges, with the first slot corresponding to the Hexagon sharing the vertex on slot 1.

2.1.5 Players Structure

A Player structure shall store information relating to the gameplay of a particular player. This information shall be used for accounting and display purposes only. However, the global game status is stored by the global game engine. In the case of a mismatch between the information stored by Player and the one by Board, the latter shall take precedence.
Players shall have:

- A HashMap of UserResources, an encapsulated type that references to a portion of the game resources Board has registered.
- An ArrayList of UserElements, an encapsulated type that references to a type of element (settlement, road, army, etc.), an argument (e.g., type of settlement) and a reference to the appropriate Vertex/Edge.

AIPlayer Structure

The AIPlayer structure shall be used for gaming conducted by an artificial intelligence (AI) engine. This engine shall be backed by a checkboard. On top of the basic Player structures, an AIPlayer shall operate using different TreeMap structures, assigining points based on the type of action and the location available. These structures offer the main advantage that they are sorted and thus result optimal for “best decision” algorithms.

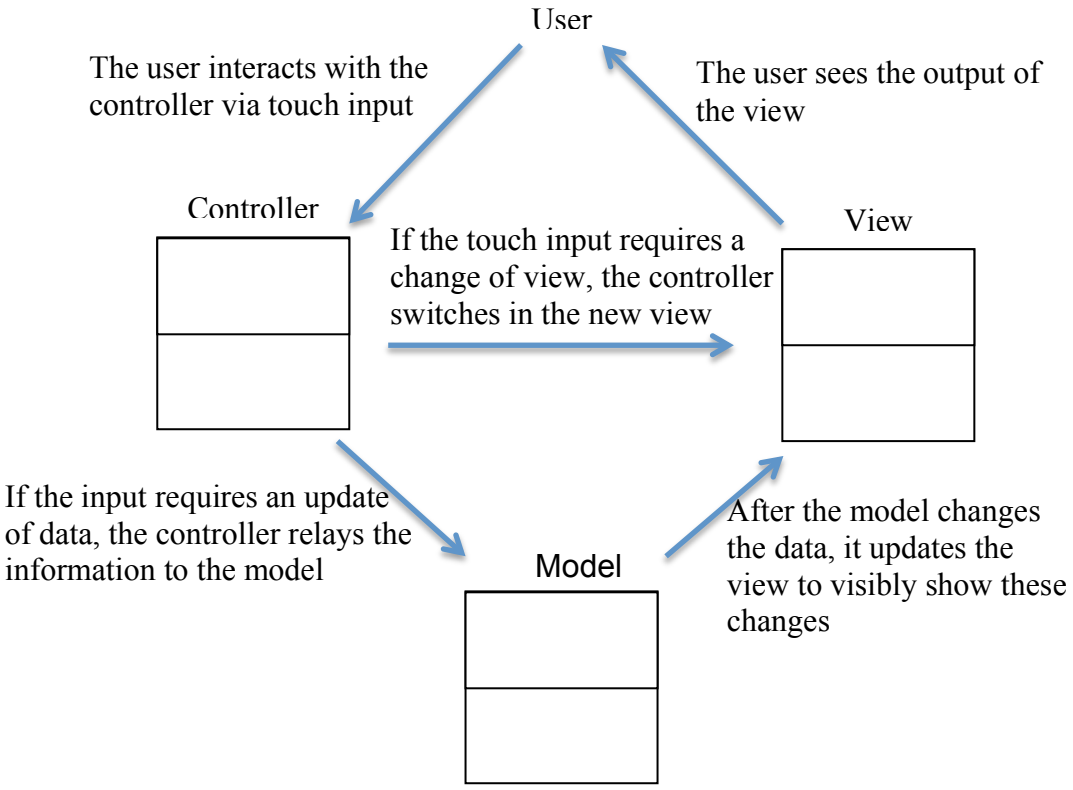
2.2 Temporary data structure

The gameplay is conducted mainly on a specific session, and therefore most structures are temporary by definition. However, in certain cases some temporary files may used for error recovery purposes. These structures shall be implemented as part of the Serializable interface.

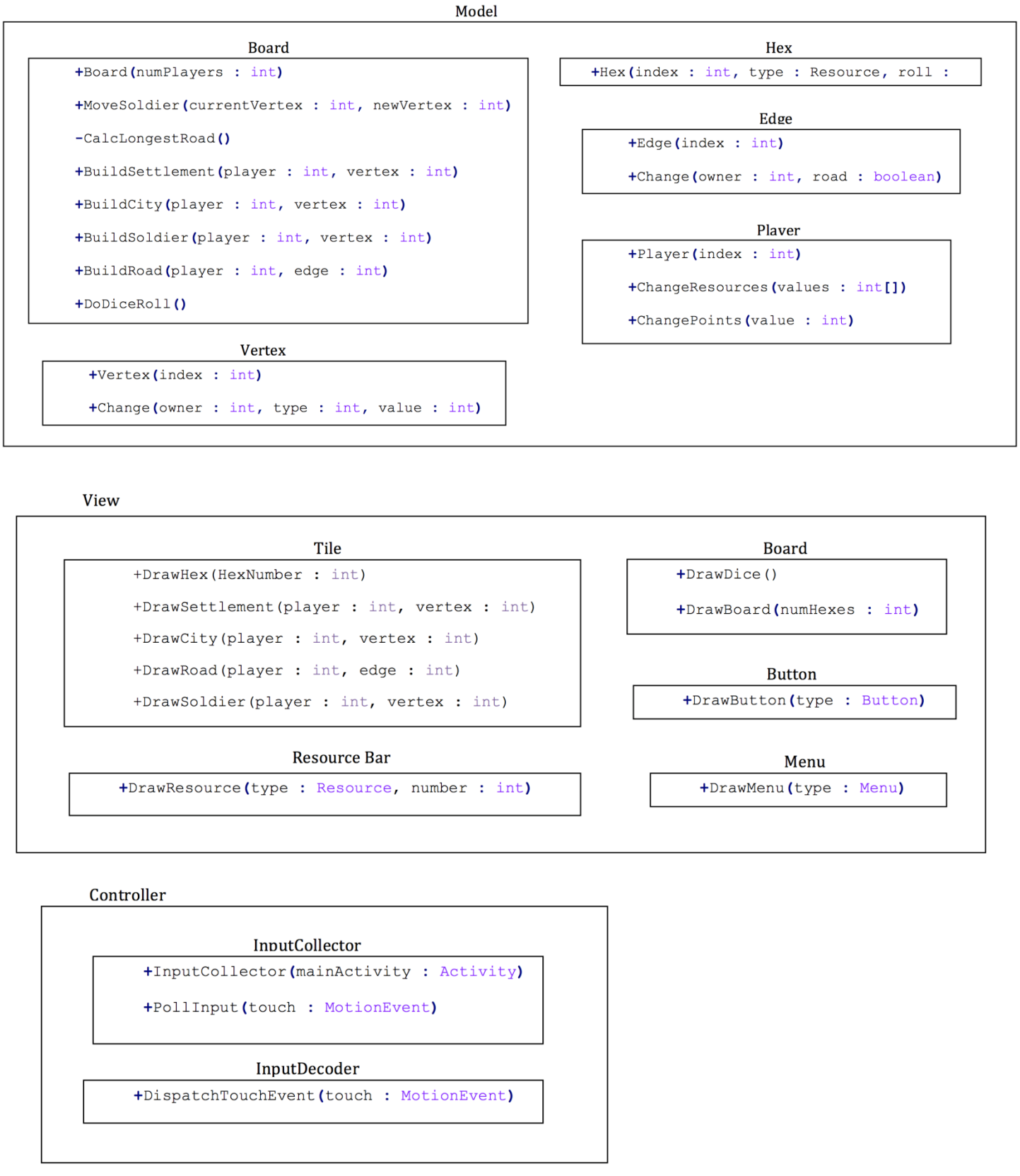
3 Architectural and component-level design

3.1 System Structure

Risky Business uses a model-view-controller system to accept user input, carry out operations, and display results. Everything is built from the ADK. The view is in charge of displaying all objects to the screen. It will render the game board, and menus. The controller will handle user input. Once input is accepted, it will determine what action needs to be taken. The model handles the backend. The underlying game board structure and methods are all present here. When the game board changes data, it will update the view to render those changes. This can be seen visually by this flowchart:



3.1.1 Architecture diagram



3.2 Description for Controller

The controller provides for communication between the user and the model. The controller is the part of the program that will register all user interactions, and then tell the model what to do based on the user inputs. The controller will also select different views.

3.2.1 Processing narrative (PSPEC) for Controller

The controller is called upon every user interaction. If the user is not interacting, then the controller will be polling for input. Upon user interaction, the controller will carry out the action defined by the user input. The controller is in charge of selecting the view (changing menu screens, going from splash screen to main menu, opening up the build menu, etc.), as well as calling the model based on user input. Ex: The user taps on the build settlement icon. It is the controller's job to accept the tap, determine where the tap occurred, and call on the Boards build settlement method. The Model will then take care of the rest. In this case the controller does NOT tell the view that a settlement is built, that is the Board's job.3.2.2 Component "n" interface description.

3.2.2 Controller Interface Description

All touches to the screen are caught by the controller. Once the controller accepts a touch, it will determine the touches location and meaning, and call the proper method from either the Model or View.

3.2.3 Controller processing detail

The controller is composed of two components: Input Collection, and Input Decoding.

Input Collection: The controller will poll for user input. The majority of the input will be taps, so all other inputs will be thrown away

Input Decoding: The controller will take the user input, and determine what the user clicked on. It will then relay this information to the appropriate class and method.

3.2.3.1 Design Class hierarchy for controller

The controller has only two classes, with no hierarchy. Its job is to accept and process user input.

3.2.3.2 Restrictions/limitations for controller

At this time the controller will accept user input, and process it. The processing of the user input may not be able to adequately be accomplished by the controller. This may have to be passed on to a separate component. The limitation here is that although the

controller knows where the touch is received, it is unaware of what is located at that touch point.

3.2.3.3 Performance issues for controller

No perceived performance issue. The controller will poll for input, and process it.

3.2.3.4 Design constraints for controller

The controller has limited knowledge. It can accept input, and determine where the input came from, but it may or may not be able to determine the intent of that input.

3.3 Dynamic Behavior for Controller

The controller is the only component that interacts with the user. It will take user input, and then relay this information to the appropriate class.

3.3.1 Interaction Diagrams

A sequence diagram, for each use case the component realizes, is presented.

3.4 Description for Model Component

The model component manages data relating to the current game state. This includes the layout of the hex for the current game, the player resources, and the units on the board.

3.4.1 Processing Narrative (PSPEC) for Model Component

The initial model will be generated upon start-up and include the basic layout of the board. From there onwards, whenever the model updates any visible aspect of the board, it will call the appropriate method of the View component in order to relate this change to the user(s). After the board is generated, control flow will pass back to the Main component. After generation, all changes to the Model will be a result of a Controller and hence the methods within Model will only be called by a corresponding Controller method; the argument of the call will contain all necessary information regarding what the Controller wishes to change and at no point will any Model method interact directly with a user.

3.4.2 Model Interface Description

Board

```
+Board(numPlayers : int)

+MoveSoldier(currentVertex : int, newVertex : int)

-CalcLongestRoad()

+BuildSettlement(player : int, vertex : int)

+BuildCity(player : int, vertex : int)

+BuildSoldier(player : int, vertex : int)

+BuildRoad(player : int, edge : int)

+DoDiceRoll()
```

Hex

```
+Hex(index : int, type : Resource, roll : int)
```

Edge

```
+Edge(index : int)

+Change(owner : int, road : boolean)
```

Vertex

```
+Vertex(index : int)

+Change(owner : int, type : int, value : int)
```

Player

```
+Player(index : int)

+ChangeResources(values : int[])

+ChangePoints(value : int)
```

3.4.3 Board Component Processing Detail

At the beginning of the game, once the number of players is known, the constructor of `Board` will be called, which will randomly generated an appropriately sized game board. During this generation, it will construct an

`ArrayList` for each of the board components, `Hex`, `Edge`, and `Vertex`. In addition, it will create a list that maps dice rolls to the indices of the hexes that have those rolls. This greatly decreases look-up time during the dice roll and harvest phase. These lists will completely encapsulate all the necessary information to construct the visual representation of the board. In addition, there will be an object of a class that inherits `Player` for each player, including both human users and AIs. There will be a sub-class for each of local players, online players, and computers. These sub-classes will contain the necessary information, such as resource amount, to generate the visual representation both for that player and what the other players will see, such as the number of cards they have. After the initial constructor, the Controller component will initiate calls to the methods in the Board component always, as the game state only changes as a direct result of a player performing some action.

3.4.3.1 Design Class Hierarchy for Board Component

As above, there will be three sub-class of `Player`, which is an abstract class, for each type of player. Other than that, the classes have no hierarchy.

3.4.3.2 Restriction/Limitations for Board Component

Ideally, there will be no restrictions on the inputs. Instead, the Controller component should check that the user's input is valid before making a call to the Board component.

3.4.3.3 Performance Issues for Board Component

All changes to the game state should be done quickly enough that the user experiences no lag time. Because the size of the data structures encapsulating the board are relatively small, this is not an issue. Each list will not exceed more than a few hundred elements, as the board will have under a hundred hexes.

3.4.3.4 Design Constraints for Board Component

No method shall directly interact with a user. The Controller component will process the input and pass all the necessary information to the Board component.

3.4.3.5 Processing Detail for each Operation of Board Component

- `Board(numPlayers : int)` will choose the board shape, generate random resource and dice rolls for each hex, and then initialize and instantiate the lists describing the board.

- `MoveSoldier(currentVertex : int, newVertex : int)` will change the appropriate vertices, as well as check if any roads are affected. If so, they are also updated.
- `DoDiceRoll()` will find for the hexes that match the dice roll and then check for adjacent buildings, at which point it will update player resources.
- `ChangeResources(values : int[])` will take an array with one element for each resource and then update the player resources by those amounts. The amounts can have any sign.

3.2.3.5.1 Processing Narrative (PSPEC) for each operation

- `Board(numPlayers : int)`: creates board.
- `MoveSoldier(currentVertex : int, newVertex : int)` moves soldier.
- `DoDiceRoll()` harvests resources.
- `ChangeResources(values : int[])` update player's resources.

3.4.3.5.2 Algorithmic Model for each operation

- `Board(numPlayers : int)`:
  - Compute board size and layout.
  - Generate dice rolls and resources for hexes.
  - Construct lists representing the board layout.
  - Call to View component.
- `MoveSoldier(currentVertex : int, newVertex : int)` moves soldier.
  - Checks for road between the two vertices.
  - Updates road if necessary.
  - Changes soldier location.
  - Call to View component.
- `DoDiceRoll()` harvests resources.
  - Generate dice roll.
  - Look-up affected hexes.
  - Check surrounding vertices for buildings.
  - Update player resources.
  - Call to View component.
- `ChangeResources(values : int[])` update player's resources.
  - Change each resource accordingly.
  - Call to View component.

3.5 Dynamic Behavior for Board Component

Calls to this component are a direct result of operations in the Controller

component. The board is generated once the game is started. Subsequently, each call to this component comes from Controller. After performing an operation in this component, the View component will always need to be updated.

3.6 Description of View

The View component is responsible for displaying the output of data through the use of a user interface (UI). It requests data from the Model component and displays it to the user in a clear and understandable way.

3.6.1 Processing Narrative (PSPEC) for View

When a new game is begun, the game board will be drawn The View component is called whenever the Model component makes changes to the board or to the user's data (for example, resources). These changes are displayed to the user in the form of a user interface. The user will then be able to make decisions based on the output; if the output is a building screen, the user may select one of the options. Their selection will be handled by the Controller. This selection can be passed by the Controller to the View if the user's selection is dealing with navigation through menus.

3.6.2 View Interface Description

The View component is implemented using a number of methods that will display the data within the Model component. It will accept no input from the user.

Tile

```
+DrawHex(HexNumber : int)

+DrawSettlement(player : int, vertex : int)

+DrawCity(player : int, vertex : int)

+DrawRoad(player : int, edge : int)

+DrawSoldier(player : int, vertex : int)
```

Resource Bar

```
+DrawResource(type : Resource, number : int)
```

Button

```
+DrawButton(type : Button)
```

Board

```
+DrawDice()

+DrawBoard(numbHexes : int)
```

Menu

```
+DrawMenu(type: Menu)
```

3.6.3 View Component Processing Detail

There are a number of components that compose the View:

Tile: This component draws the hexagonal tile, as well as all of the structures that are built on/around it (roads, cities, settlements, and armies)
Resource Bar: This component draws the resource bar on the game board screen
Button: Draws the buttons that the user can interact with.
Board: Draws the game board, as well as the dice on the game board screen when the dice are rolled.
Menu: Draws the menus (main, pause, trade, etc.)

3.6.3.1 Design Class hierarchy for View Component

The View component is an abstract class that is the basis for the other components that provide output to the user.

3.6.3.2 Restriction/Limitations for View Component

The View should not be able to accept any input from the user. Its job is to output information to the user, mainly in the form of the varying menus and the game board.

3.6.3.3 Performance Issues for View Component

The device that is used could potentially affect how items are displayed on screen. If the recommended type of device (tablets/phablets) is not used, the resolution and positioning of some in-game elements may be of lower quality or

improperly positioned, respectively.

3.6.3.4 Design Constraints for View Component

The View cannot pass input to the Model; this is handled by the Controller. The View will receive data from the Model component and Controller.

3.2.3.5.1 Algorithmic Model for each Operation of View

Tile

DrawHex: Given a number from the list of hexagonal tiles present
DrawSettlement: Draws the settlement on the appropriate vertex.
DrawCity: Draws a city that the user builds; places it on the selected vertex of a tile.
DrawRoad: Draws a road that the user builds; places it on the selected edge of a tile.
DrawSoldier: Draws a soldier that the user purchased; places it on the selected edge of a tile.

Resource Bar

DrawResource: Draws the resources on the game board screen for the user. This is called when an update is made within the Model component.

Button

DrawButton: draws a button depending on what menu is open. The building screen will have a button that says “Build”.

Board

DrawDice: draws a pair of dice on the gameboard screen when they are rolled.
DrawBoard: creates the initial board at the beginning of the game. The number of tiles will vary based on the input.

Menu

DrawMenu: draws the appropriate menu depending on the gamestate.

3.7 Dynamic Behavior for View Component

Since the View component is dependent on the Model component, when an operation is performed in the latter, there will be a change made in the former. When pertaining to menus, options will be made available to the user; their selections will be handled by the Controller. The View can take input from the Controller, in the situation that the user is navigating through menus.

4 User interface design

The user interface consists of six main components: A main game menu, options menu, gameplay screen, pause menu, item trading screen, and high score viewing screen.

4.1 Description of the user interface

Main Menu: The main game menu will be the first screen the user sees as the game starts. This menu gives the user options to start a new game, load a previously saved game, view current high scores, access to the options menu, or exit the application. The high scores option will take the user to a leaderboard of current high scores of the game.

Options Menu: If the options menu is selected it will pull up a new screen with difficulty selection, avatar selection, turning game sounds on/off, view the tutorial, and a resume button indicating return to the main menu.

Gameplay Screen: The gameplay screen will be primarily the Settlers of Catan game board with user interaction capabilities. The user interactions that pertain to gameplay include rolling dice, placing roads and settlements, utilizing armed forces to take another player's property, and trading. This will all be implemented with pop up windows. The other features that will appear in the gameplay screen include the pause button, save game option, and back to main menu.

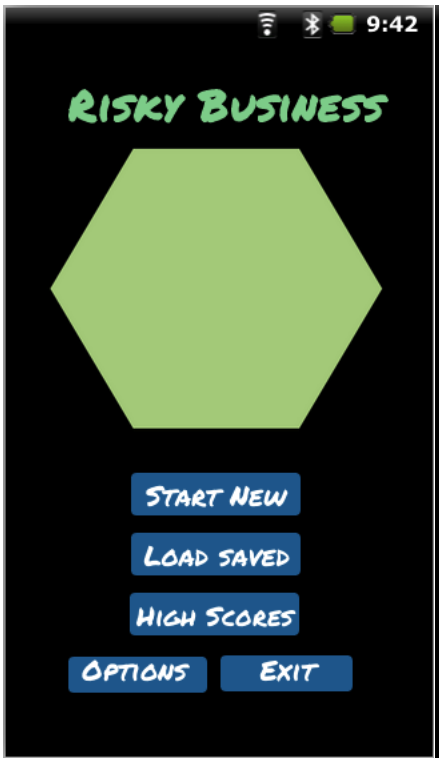
Pause Screen: The pause screen is shown when the user selects Pause in game. The pause menu has one resume game button. The player can access this pause screen at any time during the game.

Item Trading Screen: This screen shows up when the user chooses to trade with another player. This screen shows the items that are to be traded and whether the user accepts or cancels the trade.

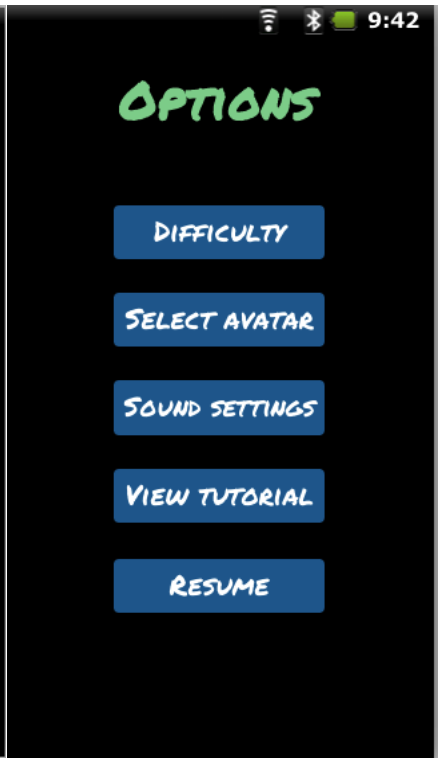
High Scores: This screen displays the high scores of players. There is a back arrow to get back to the previous screen. The high scores can be entered by the user once a game is won. The user can enter a string that is up to 15 characters.

4.1.1 Screen images

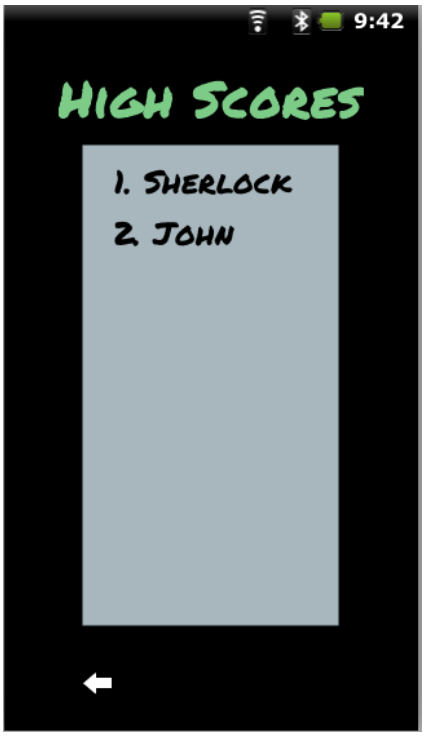
**Note: These are mockup images of the potential UI with limited functionality and assets.*



Game Main Menu



Options Menu



High Score Board

Potential Game Board

4.1.2 Objects and actions

Each of the screen on the interface will consist of various GUI elements such as PushButtons, TextEdits, and RadioButtons. The user actions will be touch based and handled by the Android device. The UI should allow the player to effectively navigate the game.

The various menus presented above provides a solid foundation to each screen and menu along with labeled buttons and stylized layouts to give a brief overview of the game UI.

4.2 Interface design rules

Ben Shneiderman's "Eight Golden Rules of Interface Design" provide the most clear and concise set of standards to use in developing the application interface. The eight rules are as follows:

1. Strive for consistency.
Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent commands should be employed throughout.

2. Enable frequent users to use shortcuts.
As the frequency of use increases, so do the user's desires to reduce the number of interactions and to increase the pace of interaction. Abbreviations, function keys, hidden commands, and macrofacilities are very helpful to an expert user.

3. Offer informative feedback.
For every operator action, there should be some system feedback. For frequent and minor actions, the response can be modest, while for infrequent and major actions, the response should be more substantial.

4. Design dialog to yield closure.
Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and an indication that the way is clear to prepare for the next group of actions.

5. Offer simple error handling.
As much as possible, design the system so the user cannot make a serious error. If an error is made, the system should be able to

detect the error and offer simple, comprehensible mechanisms for handling the error.

6. Permit easy reversal of actions.
This feature relieves anxiety, since the user knows that errors can be undone; it thus encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.

7. Support internal locus of control.
Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.

8. Reduce short-term memory load.
The limitation of human information processing in short-term memory requires that displays be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions

4.3 Components available

The user interface will consist of the following components:

Screen class:

MenuScreen

Consists of PushButtons and RadioButtons that have the following options. Start New, Load saved, High Scores, and Options. When the buttons are touched on the Android touch screen a MotionEvent is triggered. The position is calculated and if it is within the borders of the button the action is performed.

GameScreen

Consists of the game board that will utilize classes described in Section 3. There will be PushButtons and RadioButtons that have functions such as roll dice, place settlement, use soldier, trading, pause, save game, back to Main menu. When buttons are pressed the corresponding MotionEvent is triggered.

PauseScreen

Consists of an image and PushButton that allows you to go back to the

Main Menu.

TradingScreen

When the user initiates a trade this menu will pop up with the trading items and 2 PushButtons that will confirm or deny the trade. The buttons will have a MotionEvent attached to the corresponding action.

4.4 UIDS description

Photoshop or other image manipulation software may be used for layout purposes.

5 Restrictions, limitations, and constraints

The development of our game will be limited or restricted by the following factors:

Our greatest limiting factor with our software is time. The development team can only do so much with the time that is given to us and thus we must scale back on some of the features and/or options we had originally come up with for the SRS document, such as mini-games or fully functioning artificial intelligence. However, there are some features we would like to develop if there is sufficient time allotted, such as playing with people over wireless, pinch-zoom, or a scripted demo mode. Implementing these would be contingent on how early we are able to make the core functionality work properly.

Our software will run exclusively on devices running the Android OS. There will not be an iOS or Windows Phone equivalent developed. The software will not interact with any other third-party applications running concurrently on the device.

The source code for our software will follow the principles of object oriented design (OOP) – Encapsulation, Data Abstraction, Polymorphism, and Inheritance. The code will be written in Java and developed in Android Studio. GIT will be used for version control during the project.

Our development team is limited in terms of the number of Android devices we have readily available to directly test our software on. Due to this, many of our team's software developers will rely on emulators running virtual devices to execute Android source code.

Ideally, our software will use less than 50 MB of RAM at peak consumption, with the expectation of significantly less average RAM utilization. Additionally, the game will not prevent incoming alerts by overdrawing device resources, with such a restriction the primary bound for the amount of processing power consumed.

We will assume that all devices running our software will use the latest version of the Android OS (v4.4.2, a.k.a "KitKat"). Ideally, the software will be stable if a new iteration of the Android OS becomes available during the development process (v4.5).

Our software depends on the OpenGL ES libraries for displaying the game on the screen. Due to this, if the libraries are suddenly no longer supported by Android (i.e. a new iteration of the Android OS becomes available), system failures would be expected to happen while attempting to run the game.

Since our development team and target audience are either fluent in English or English is their first language, all text and documentation for Risky Business will done in American English. Therefore, no foreign language versions will be developed.

After development on the game has wrapped at the end of the semester, there is no guarantee of any revisions, updates, or maintenance to the software by the development team.

6 Testing Issues

Risky Business will be tested in a variety of ways to ensure complete functionality of the overall program and its individual components. Testing will be done on an individual class basis before testing of the complete program occurs. Gameplay will be tested through player versus player testing as well as player versus AI.

6.1 Classes of tests

6.1.1 General System Testing

Program features will all be tested individually before being combined and tested together. Individual features include but are not limited to the options menu, game board, trade and building screens, and general gameplay in each play mode. The initial testing of these features will be done utilizing white box testing (full or partial knowledge of the internal specification). Once combined the complete program will undergo testing in both white box and black box (no knowledge of the internal specification) fashions.

6.1.2 Gameplay Testing

Once program functionality is established, testing of Risky Business's gameplay will need to be tested. Each method of gameplay will have to undergo both white box and black box testing.

6.1.2.1 Player vs Player Gameplay

Risky Business will offer a pass and play gameplay option. Two users will have to be present to complete this test. The two of them will attempt to play through the game normally, along the way they will attempt to "break" the game by giving input that might yield errors. Users testing the game will need to fall under both white box and black box categories to ensure a complete examination of the program.

6.1.2.2 Player vs AI

Risky Business will offer a user vs AI gameplay experience for tutorials. Additionally the Player vs AI mode will aide in testing general system features without the requirement of two users. The

AI will need to conduct basic gameplay moves and make basic, but logical, gameplay decisions. The AI should not be difficult to beat but will serve as a way of learning to play the game. Player vs AI testing will be conducted in the same manner as Player vs Player in regards to the white box and black box requirements.

6.1.3 Screen Testing

Once integrated as a complete program, screen testing will be required to see if the different game and menu screens interact with each other correctly. Screen testing will ensure that opening multiple overlays over the main game board screen will not interfere with anything. The screens that must be tested are the previously mentioned game board, main menu, options menu, trade/building menu, and various pop-up messages.

6.2 Expected software response

6.2.1 Gameplay

Testing will ensure that gameplay in any mode works without glitch and functions until completion.

6.2.1.1 Player vs Player

The player vs player pass and play game mode will work the same for each participant and provide an equal opportunity for each player to win.

6.2.1.2 Player vs AI

The player vs AI game mode is expected to provide a game tutorial for the user. This game mode is expected to conduct a complete game between the user and the AI.

6.2.2 Screen Transitions

Testing will ensure that each individual screen of the program will function completely and also not interfere with other screens. The program will be able to transition smoothly between the main menu and into its sub categories. The game board screen will change as the game progresses to show the actions made by the players. This screen is expected to update at the end of each turn and always remain accurate.

6.3 Performance bounds

Risky Business is a turned based game however it is expected to update in real time. Once moves are made by the players it is expected that the game board will update accordingly and without hesitation. The program will load in a timely fashion to ensure easy pick up and play. The program will transition between its different screens smoothly and will not incur graphical errors when multiple screens overlay on each other. Risky Business is expected to be played on an Android Tablet system and it will not require additional memory or performance requirements.

6.4 Identification of critical components

To ensure basic gameplay functionality, Risky Business requires a working game board and trade/building screen. Also it is essential that the game supports multiple players.

6.4.1 Critical System Features

As previously stated the most critical program features are the features required for gameplay. These features include a game board that updates and reflects the current state of the game to the users, as well as a trading/building screen that players will utilize to progress the game along. Additionally the program must allow for at least a two player PvP gameplay mode. Since Risky Business is a two player game it is essential that program accepts commands from two users.

7 Appendices

7.1 Requirements Traceability Matrix

	View Component	Touch Input	Sound	Score Saving	UI	Resources/ Data
Controller	x	x		x	x	
Screen	x					
Textures	x					
MainScreen	x	x	x		x	
GameScreen	x	x	x		x	
BuildScreen	x	x	x		x	
TradeScreen	x	x			x	
CreditsScreen	x	x			x	
SettingsScreen	x	x	x		x	
SetupScreen	x	x			x	
SplashScreen	x				x	
Hextile	x					x
Board	x					
Edge	x					
Vertex	x					
Dice	x		x			
Army	x					x
Resources						x
ResourceBar	x				x	x

Player				x		x
PushButton	x	x			x	
RadioButton	x	x			x	
Slider	x	x			x	

7.2 Packaging and Installation Issues

The Risky Business Android application will be packaged as an APK file and installed onto the tablet from a computer through USB connection.