University of Idaho
Open Space. Open Minds.

Swords and Sorcery Computer Adaptation Documentation
An overview of requirements, design, implementation, testing and metrics
Version 1.0

Prepared by:
University of Idaho Computer Science 383 Class, Spring 2014

Prepared for:
Dr. Clinton Jeffery

May 9, 2014

Introduction and Document Description

This is a collection of the documentation gathered for the computer adaptation of the board game, Swords and Sorcery. The software and system for the computer adaptation of S&S was implemented in Software Engineering, Spring of 2014, at the University of Idaho. Included are the following documents, in their entirety:

- Systems and Software Requirements Specification

- System and Software Design Description

- Implementation Description

- Testing Description

- Metrics Description

SYSTEMS AND SOFTWARE REQUIREMENTS SPECIFICATION (SSRS) FOR

CS383 Spring 2014 Project: Swords and Sorcery

Version 1.0
5/9/14

Prepared for:
CS 383 Software Engineering Spring 2014

Prepared by:
Software Engineering Class Spring 2014
University of Idaho
Moscow, ID 83844-1010

SWORDS AND SORCERY SSRS
TABLE OF CONTENTS

Section Page

1 Introduction

1.1 IDENTIFICATION

The system which this document applies is titled Swords and Sorcery, or S & S for short. It is currently in version 1.0.

1.2 PURPOSE

The purpose of the system under development is to create a computerized version of the board game Swords and Sorcery. While the system will be used by anyone who wants to play the game , this document is intended to be read and understood by University of Idaho computer science software designers and coders.

1.3 SCOPE

This project began development at the beginning of Spring Semester 2014. As a class we began by identifying requirements, we then moved onto a brief phase of design before jumping right into implementation. We have done some testing, but most of this project is still underdevelopment. This project is sponsored by Dr. Jeffery, and is developed by the Software Engineering class. Currently the project is able to be ran on any machine capable of running the Java Virtual Environment.

1.4 DEFINITIONS, ACRONYMS, AND ABBREVIATIONS

Term or Acronym	Definition
Alpha test	Limited release(s) to selected, outside testers
Beta test	Limited release(s) to cooperating customers wanting early access to developing systems
Final test	aka, Acceptance test, release of full functionality to customer for approval
DFD	Data Flow Diagram
SDD	Software Design Document, aka SDS, Software Design Specification
SRS	Software Requirements Specification
SSRS	System and Software Requirements Specification
S & S	Swords and Sorcery
TCP/IP	Transmission Control Protocol and Internet Protocol

1.5 OVERVIEW AND RESTRICTIONS

This document is for limited release only to UI CS personnel working on the project.

Section 2 of this document describes the system under development from a holistic point of view. Functions, characteristics, constraints, assumptions, dependencies, and overall requirements are defined from the system-level perspective.

Section 3 of this document describes the specific requirements of the system being developed. Interfaces, features, and specific requirements are enumerated and described to a degree sufficient for a knowledgeable designer or coder to begin crafting an architectural solution to the proposed system.

Section 4 provides the requirements traceability information for the project. Each feature of the system is indexed by the SSRS requirement number.

Sections 5 and up are appendices including original information and communications used to create this document.

2 OVERALL DESCRIPTION

2.1 PRODUCT PERSPECTIVE

Swords and Sworcery is an independent product and all of it's functions are completely self-contained.

2.2 PRODUCT FUNCTIONS

This project's main function is to create a computerized version of the board game Swords and Sorcery, specifically the army game. This includes the game being a multi-player game to be playable over a network. The game functions are as follows

- Create/Join a game

- Play through the different phases:

  - Movement: move units and characters on the board
  - Combat: complete combat between two different stacks of units
  - Spell: cast a spell with a character

- Communicate the results of these phases over the network to other instances of the game

- Communicate with other players through chat over the network

- End the game

2.3 USER CHARACTERISTICS

The intended users of this product are anyone who wants to play the game. The user should have an education level of at least middle school so they can understand the rules of the game. They should have some experience of the rules of Swords and Sorcery. The intended user needs no technical expertise to run this application.

2.4 CONSTRAINTS

Swords and Sorcery is constrained by the rules set by the physical board game. Also due to the networking it is constrained to following the protocols over the network. Hardware requirements are just having enough memory to hold and run Swords and Sorcery, and have the hardware necessary to run the Java Run-Time Environment.

2.5 ASSUMPTIONS AND DEPENDENCIES

We assume that the Java Run-Time Environment will be able to run Java 8 code, but if future versions of the Java Run-Time Environment do not support Java 8 code then the Swords and Sorcery code base will need to be updated accordingly.

2.6 SYSTEM LEVEL (NON-FUNCTIONAL) REQUIREMENTS

2.6.1 Site dependencies

Swords and Sorcery has very few site dependencies. It requires a mouse and keyboard for input and a monitor for output. To run Swords and Sorcery requires the Java Run Time Environment that can run Java 8 applications. Also the computer needs to have a network connection.

2.6.2 Safety, security and privacy requirements

Sword and Sorcery had no safety, security, or privacy requirements.

2.6.3 Performance requirements

Swords and Sorcery software application should be able to support up to seven users playing the same game over the network.

2.6.4 System and software quality

This project shall have the ability to perform all of the required functions with consistent results. Swords and Sorcery shall also be able to be playable on any machine capable of running Java as that is what Swords and Sorcery will run on. It shall also be able to be used by anyone capable of understanding the rules to the board game.

2.6.5 Packaging and delivery requirements

The game and all of its documentation is available at https://github.com/cjeffery/sworsorc

2.6.6 Personnel-related requirements

The system under development has no special personnel-related characteristics.

2.6.7 Training-related requirements

No training materials or expectations are tied to this project other than the limited help screens built into the software and the rule book for the board game.

2.6.8 Logistics-related requirements

The system requirements for Swords and Sorcery are any machine capable of running Java 8 run time environment.

2.6.9 Precedence and criticality of requirements

All requirements stated above have equal weight, since there are no requirements for safety, security, or privacy all of the requirements were given an equal weight.

3 SPECIFIC REQUIREMENTS

3.1 EXTERNAL INTERFACE REQUIREMENTS

3.1.1 Hardware Interfaces

Swords and Sorcery did not require any hardware interfaces.

3.1.2 Software Interfaces

Swords and Sorcery only required TCP/IP as the only software interface due to needing a network connection.

3.1.3 User Interfaces

The user interface required for Swords and Sorcery is the Java Run-Time Environment. This is a requirement resulting from the implementation not the design of the project.

3.2 SYSTEM FEATURES

3.2.1 Use Case Diagrams

A pdf of our use case diagrams is inserted starting on the following page.

CS 383 Swords & Sorcery
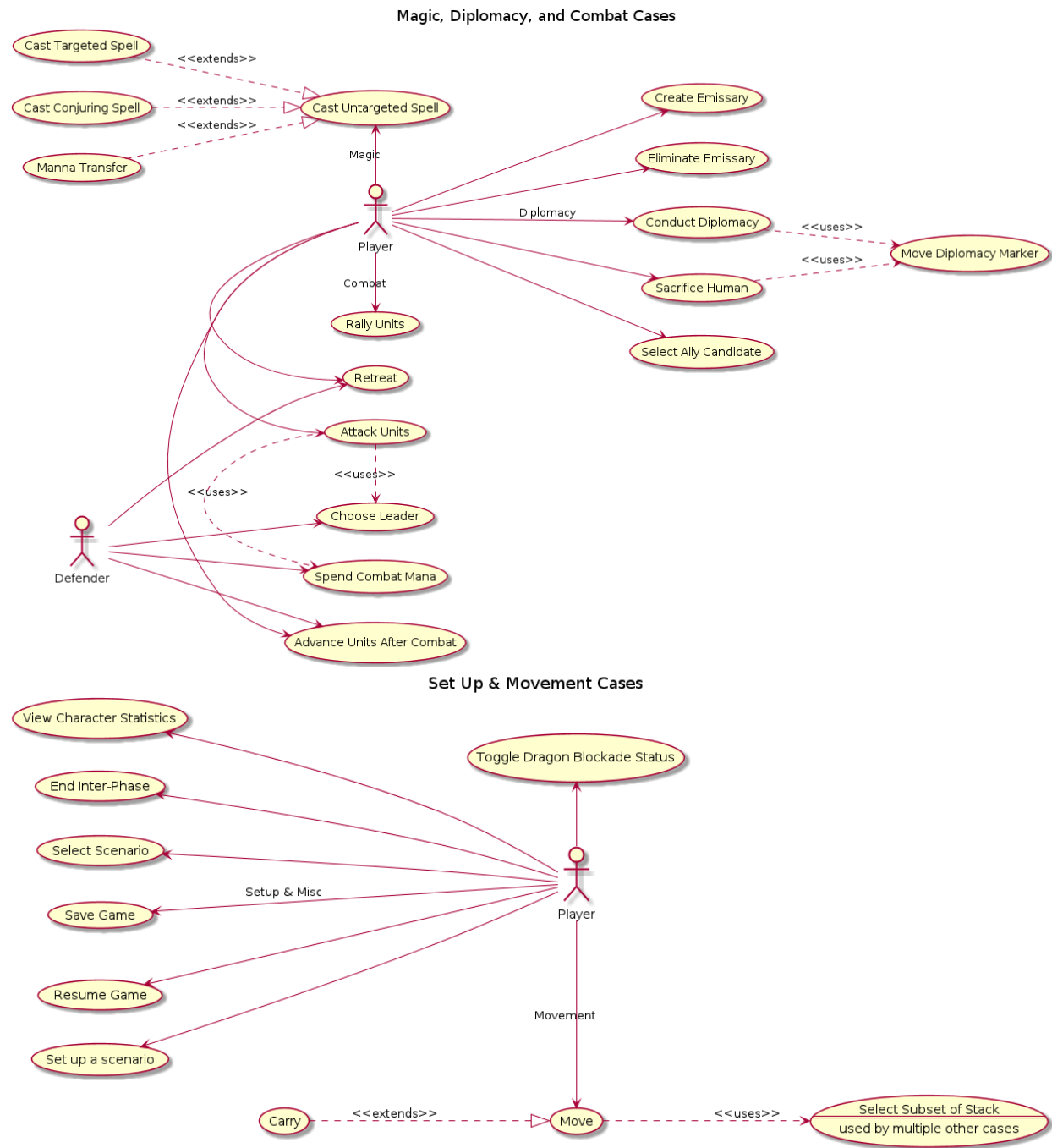# Master Use Case Descriptions & Diagram

February 18, 2014

Contents

Use Case Diagrams

Magic, Diplomacy, and Combat Cases

Cast Targeted Spell

<<extends>>

Cast Conjuring Spell

<<extends>>

<<extends>>

Manna Transfer

Cast Untargeted Spell

Magic

Player

Combat

Rally Units

Retreat

Attack Units

<<uses>>

<<uses>>

Choose Leader

Defender

Spend Combat Mana

Advance Units After Combat

Create Emissary

Eliminate Emissary

Diplomacy

Conduct Diplomacy

<<uses>>

<<uses>>

Move Diplomacy Marker

Sacrifice Human

Select Ally Candidate

Set Up & Movement Cases

View Character Statistics

Toggle Dragon Blockade Status

End Inter-Phase

Select Scenario

Setup & Misc

Player

Save Game

Resume Game

Movement

Set up a scenario

Carry

<<extends>>

Move

<<uses>>

Select Subset of Stack
used by multiple other cases

1 New Game

Actor: Player

Goal: To start a new game

Summary: Player selects options from a menu, choosing what game to initiate.

Steps:

1. Player selects "new game".

2. System displays a list of scenarios.

3. Player selects a scenario.

4. Player selects what nation they will play.

5. System adds the game to the list available in the Lobby.

6. Player proceeds to Set Up.

2 Join Game

Actor: Player

Goal: To join an available game

Summary: Player selects game from the Lobby.

Steps:

-

1. System displays a list of available games/scenario/opponent in the Lobby from main menu.

2. Player selects a game.

3. Player selects what nation they will play.

4. System displays faction information such as victory conditions and starting units.

5. Player confirms selection.

6. System informs opponent(s) of the new player.

7. Player proceeds to Set Up.

3 Set Up

Actor: Players starting a game

Goal: To do scenario set-up

Summary: Player places the allocated units in valid hexes.

Steps:

	1. System displays units to be placed and valid hexes.
-	
2. Player places units in appropriate hexes until all available units from all players have been placed.	
3. Player indicates when they are 'satisfied' with placement.	4. System toggles "satisfied" off whenever enemy units are placed.
-	5. Scenario begins whenever all players are "satisfied".

4 Resume Game

Actor: Player

Goal: To resume a saved game

Precondition: A saved game exists. The player wishes to reconnect to the existing game rather than a new one.

Summary: The player restores a previously saved state

Related:

Steps:

1. The player selects the "Resume Game" option.	2. The system displays the available saved games.
3. The player selects the desired save file.	4. The system displays a prompt.
-	5. The system loads the saved state.

5 Save Game

Actor: Player

Goal: To save the current game state for later play

Precondition: A game is currently being played

Summary: The player selects a save option and the system saves the current state.

Related:

Steps:

1. The player selects the "Save Game" option.	2. The system displays a list of available save slots.
3. The player selects the desired slot.	4. The system displays the relevant confirmation prompt.
-	5. The system saves the current state of the game.

6 Select Scenario

Actor: Hosting Player

Goal: To select a pre-made game scenario.

Precondition: A new game has been initiated

Summary: The hosting player selects a pre-made game scenario, for a new game.

Related:

Steps:

-
1. The system displays the list of available pre-made scenarios for the hosting player to select.
2. The player selects the desired scenario from the list.
3. The system displays the relevant confirmation prompt.
-
4. The system loads the selected scenario.

7 End Inter-Phase

Actor: Player

Goal: To end the current player's phase.

Precondition: The game is in a phase the player has control to end.

Summary: The player wishes to end their current phase early. If the rules are not violated the turn is ended.

Related:

Steps:

1. The player selects an "End Phase" option
2. The system displays the relevant confirmation prompt.
-
3. The system ends the current phase.

8 Select Ally Candidate

Actors: All Players

Goal: To determine player alliances for the game-turn.

Precondition: It is currently the player-Order Determination Inter-Phase.

Summary: Each player selects any other player(s) they would like to be allied with for the turn.

Steps:

-

1. The system displays to each player a list of potential allies (the other, eligible, players).

2. The players select any other player(s) they wish to ally with for the game-turn.

3. The player indicates they are finished selecting hopeful allies.

4. The system moves to the next phase.

9   Move

Actor: Player

Goal: To move unit(s) or character(s) from one tile to another

Summary: Move a unit, character, monster, stack, or subset of a stack from a starting tile to a destination tile, depending on terrain or movement points.

Precondition: It is the player's movement phase

Related: Carry, **Select Subset of Stack**

Steps:

1. The player selects a unit, character, monster, stack, or stack subset.

2. The system displays what hexes are reachable by those unit(s) based on their remaining Movement Points.

3. The player selects a destination hex to move the unit(s) to.

4. The unit(s) are moved and their movement points are subtracted accordingly.

Alternatives:

1. If unit is moved into an enemy zone of control, it's movement might immediately end.

2. If a unit movement would trigger invasion, the player is prompted to confirm the action.

3. If a unit move ends on an enemy character, the character is captured.

4. The user can cancel movement after being shown the reachable hexes.

5. Some unit(s) will not be able to make a valid movement for various reasons, in this case the user must cancel the movement.

10   Teleport

Actor: Player

Goal: To carry a character using a flying unit

Summary: Flying units or monsters can carry other characters if said other characters haven't moved yet.

Precondition: It is the movement phase and the player has a flying unit that has room on it's back, in range of a character that hasn't moved yet

Related: Movement

Steps:
1. Player moves on top of a portal hexagon.
2. Player is provided a dialog giving them the option to use the portal.
3. the player chooses to teleport his units individually or as a group.
4. Perform appropriate teleportation.

Alternatives:

1. Should an enemy unit occupy an output portal, the teleported units should be retreated by one tile.
2. A player can choose not to teleport in step 2.
3. A player may have a spell (Teleport Protection, Teleport Control) affect the outcome of step 4.

11 Carry

Actor: Player

Goal: To carry a character using a flying unit

Summary: Flying units or monsters can carry other characters if said other characters haven't moved yet.

Precondition: It is the movement phase and the player has a flying unit that has room on it's back, in range of a character that hasn't moved yet

Related: Movement

Steps:
1. The player moves a flying unit over unmoved character(s) as per **Movement**
2. The system asks which, if any, characters should be carried.
3. The player indicates which, if any, characters they want carried.
4. Those characters are stacked with the flying unit for the remainder of the movement phase.

12 Select Subset of Stack

Actor: Player

Goal: To select one or more units out of a stack

Summary: There's lots of unit stacking in Swords & Sorcery. The player needs an easy way to both use a stack as a whole or to use specific units from a stack.

Steps:
1. The user selects a stack.
2. The user is given the choices of using the stack as a whole, selecting one or more units out of the stack, or cancelling the selection.
3. The user either cancels or selects the desired unit(s).
4. Those units become selected.
5. The user continues on with whatever action they were going to do using the selected units.

Alternatives: 1. In some states it might only make sense to select a single unit out of a stack. In this case the user is not given the choice of selecting multiple units or the whole stack

2. In some states such as **Eliminate Unit** the player may be forced to select some number of units. In this case there is no option to cancel

13   Attack Units

Actor: Player

Goal: Attack opposing player through combat

Precondition: Player is in the Combat Resolution Segment

Summary: Player selects units to attack and unit to be attacked. Combat is resolved by the system.

Related: Choose Leader, **Spend Combat Manna**

Steps:
1. Attacker selects units to attack.
2. Attacker selects unit to be attacked.   3. System notifies defending player.
4. If multiple leaders (Choose Leader).
5. If magic capable leaders (Spend Combat   6. System displays effects of combat.
   Manna).

Alternatives: 1. User deselects units in step 1.

2. Combat is not valid in step 2.

14   Advance Units After Combat

Actor: Player

Goal: To move units forward into the enemies retreating path

Precondition: Player has won combat and opposing player has retreated units.

Summary: Player can choose to move units into retreating path of enemy after combat.

Steps:
1. Player selects units to advance.
2. Player selects path of advance.   3. System checks if path is valid and displays result.

Alternatives: 1. Player chooses not to move.

2. Player deselects units in step 1.

3. Path is not valid so unit does not move.

4. If a unit advance ends on an enemy character, the character is captured.

15 Retreat

Actor: Player

Goal: To retreat units after losing combat

Precondition: Player has just lost combat

Summary: The player can choose to retreat units after combat or kill off units.

Steps:
1. System shows player units to be retreated.

2. Player selects units to kill.

3. Player chooses path for remaining units.   4. System checks if path is valid and displays results.

Alternatives: 1. Units are all killed in combat.
2. Player kills all units in step 2.
3. Path not valid in step 3

16 Choose Leader

Actor: Player

Goal: To select one leader if multiple are present

Precondition: Player has more than one leader.

Summary: Player chooses which leader to use.

Steps:
1. Player selects leader from stack.   2. System shows player which leader is to be used.

Alternative: Player deselects unit in step 1. Leader is not allowed to be used.

17 Spend Combat Manna

Actor: Player

Goal: Cast magic from leaders in combat

Precondition: Player is in combat and has magical leader with manna points

Summary: Player chooses which leader to use.

Steps:
1. Player selects character.

2. Player chooses amount of manna to spend.   3. System shows player combat strength of spell.

Alternatives: 1. Player deselects unit in step 1.
2. Player does not have enough manna in step 2.

18 Rally Units

Actor: Player

Goal: Attempt to rally units that have been demoralized

Precondition: Player has demoralized units and leader is present in stack. player in Unit Rallying Segment of Army Combat Phase

Summary: Player chooses leader to use if multiple and is shown if rally was passed.

Steps:
1. Player selects demoralized units to be rallied.
2. Player selects leader to rally with, if there are more than one.
3. System shows result of rally.

Alternatives: 1. Player deselects units in step 1.

2. Leader is not allowed to be used.

3. Units cannot be rallied.

19 View Character/Unit Statistics

Actors: Player

Goal: view the statistics and current state of a certain character, unit, or monster in play.

Precondition: a character, unit, or monster is in play.

Summary: The user performs this action when he wishes to view the full statistics relating to a character, unit, or monster in play. It is similar to consulting the characters card in the manual system.

Steps:
1. Player selects the character, unit, or monster
2. System displays a menu asking what the user wants to do with the selected object (may include Move, Attack, Cast spell, View, etc. . . depending on game phase).
3. Player chooses View.
4. System displays the character, unit or monsters relevant statistics (those on the game card plus manna level and movement points remaining if applicable) in a new window
5. User closes the window.

Alternative: User can choose Cancel instead in Step 3 to close the menu.

20 Cast Untargeted Spell

Actors: Player

Goal: To cast a particular spell on a particular target, which may be a group of hexes, hex sides, characters, or units.

Precondition: Player fulfills a set of preconditions for at least one spell. These preconditions depend on the characters available, their manna levels, and the game phase.

Summary: The actor will choose one of his characters, and select a spell.

Steps:

1. Player selects one of his or her characters with a mouse click

2. System displays a menu asking what the user wants to do with the character (may include Move, Attack, Cast spell, View, etc. . . depending on game phase).

3. Player chooses Cast spell.

4. System displays a window listing all spells that the character is capable of attempting given its Magic Power Level, current Manna points, the game phase, which types of spells the character has cast that turn, and how many game-turns are left in the game play.

5. User selects one of the untargetted spells.

6. System calculates the result if the character was attempting a higher order spell, then makes internal adjustments to create the force walls.

-

7. System reports the spells results in a dialog box.

8. User accepts results, closing dialog box.

Alternative: User may choose to cancel in steps 3, or 5.

21   Cast Targeted Spell

Actors: Player

Goal: To cast a particular spell on a particular target, which may be a group of hexes, hex sides, characters, or units.

Precondition: Player fulfills a set of preconditions for at least one spell. These preconditions depend on the characters available, their manna levels, the targets available, and the game phase.

Summary: The actor will choose one of his characters, select a spell, and then choose one or more targets based on the cost of the spell.

Related: Uses "Select Subset of Stack" and extends "Cast Untargeted Spell".

Steps:

1. Player selects one of his or her characters with a mouse click

2. System displays a menu asking what the user wants to do with the character (may include Move, Attack, Cast spell, View, etc. . . depending on game phase).

3. Player chooses Cast spell.

4. System displays a window listing all spells that the character is capable of attempting given its Magic Power Level, current Manna points, the game phase, which types of spells the character has cast that turn, and how many game-turns are left in the game play.

5. User selects one of the targetted spells.

6. System highlights possible target hex sides and asks the user to choose one.

7. User selects up to as many of the specific type of target as his character can afford to cast the spell on at a cost of 2 Manna points per hex side. When done, user can press continue.

8. System calculates the result if the character was attempting a higher order spell, then makes internal adjustments to create the force walls.

-

9. System reports the spells results in a dialog box.

10. User accepts results, closing dialog box.

Alternative: User may choose to cancel in steps 3, 5, or 7.

22   Cast Conjuring Spell

Actors: Player

Goal: To conjure a particular type of unit to aide the player.

Precondition: It must be the player's Magic phase. The user must have a character of attempting a conjuring spell and have Manna points for it. No non-conjuring spells may have been cast by this character this game-turn.

Summary: The actor will choose one of his characters, select a spell, and then choose how long the unit is to remain in play.

Related: Extends "Cast Untargeted Spell".

Steps:

1. Player selects one of his or her characters with a mouse click	2. System displays a menu asking what the user wants to do with the character (may include Move, Attack, Cast spell, View, etc. . . depending on game phase).
3. Player chooses Cast spell.	4. System displays a window listing all spells that the character is capable of attempting given its Magic Power Level, current Manna points, the game phase, which types of spells the character has cast that turn, and how many game-turns are left in the game play.
5. User selects one of the targetted spells.	6. System highlights possible target hex sides and asks the user to choose one.
7. User selects up to as many game-turns as he can afford, given the cost of the spell.	8. System calculates the result if the character was attempting a higher order spell, then makes internal adjustments to create the force walls.
-	9. System reports the spells results in a dialog box.
10. User accepts results, closing dialog box.	

Alternative: User may choose to cancel in steps 3, 5, or 7.

23 Cast CounterSpell

Actors: Player

Goal: To negate the effects of certain spells

Precondition: It must be an opponent player's CounterSpell segment. The opponent must have just cast a counter-able spell: Fear, etc. The Player must have a magic character with a level high enough to cast the appropriate counter spell.

Summary: All Opponent players may choose to cast counterspells, without knowing whether each other are doing so [e.g. chat cut-off].

Steps:

-	1. System displays a dialog informing the user that so-and-so has cast the such-and-such spell, and asks if they want to counter it.
2. Player chooses Cast counter spell.	3. System displays a window with the outcome.

24 Manna Transfer

Actors: Player

Goal: To transfer Manna between a player's characters.

Precondition: It must be this players Magic phase, and the user must have a character capable of attempting a rst level spell, and this character must have 2 Manna points. Another of the players magic-capable characters must reside in the same hex as the casting character.

Summary: the user selects a character, chooses this spell, and then selects how much Manna to transfer and to what character.

Related: Extends "Cast Untargeted Spell".

Steps:

1. Player selects one of his or her characters

2. System displays a menu asking what the user wants to do with the character (may include Move, Attack, Cast spell, View, etc. . . depending on game phase).

3. Player chooses Cast spell.

4. System displays a window listing all spells that the character is capable of attempting given its Magic Power Level, current Manna points, the game phase, which types of spells the character has cast that turn, and how many game-turns are left in the game play.

5. User selects Manna Transfer.

6. System asks user to select a character to transfer Manna to.

7. User selects one of his characters in the same hex as the casting character.

8. System asks how much Manna is to be transfered.

9. User enters a number no greater than half of the casting character's current Manna level and no greater than the receiving character's maximum Manna level.

10. System calculates the result if the character was attempting a higher order spell, then makes internal adjustments to transfer the Manna.

-

11. System reports the spell's results in a dialog box.

12. User accepts results, closing dialog box.

Alternative: User may choose to cancel in steps 3, 5, 7, or 9.

25 Create Emissary

Actor: Player

Preconditions: Friendly movement phase; character with diplomatic rating greater than 0; character has 1 or 0 emissaries.

Summary: The player creates an emissary

Steps:

1. User selects character which will spawn emissary.

2. System creates the emissary and updates the board.

26 Eliminate Emissary

Actor: Player A, Player B

Precondition: Emissary occupies same hex as enemy unit(s).

Summary: One player eliminates another players emissary.

Steps:

1. Player A indicates they wish to terminate Player B's emissary.

2. System removes emissary and updates the game board.

27 Conduct Diplomacy

Actor: Player

Precondition: Player controls an emissary or character that occupies a neutral Capital hex.

Summary: Player conducts diplomacy using a character or emissary at a neutral Capital.

Related: Move Diplomacy Marker

1. User chooses to conduct diplomacy.

2. System indicates which emissary or character pieces are eligible.

3. User selects emissary or character.

4. System calculates results of diplomacy.

5. User moves diplomacy marker as in **Move Diplomacy Marker**

Alternative: System calculates a negative result for diplomacy and moves the diplomacy marker itself.

28 Move Diplomacy Marker

Actor: Player

Goal: Move a player's diplomacy marker.

Precondition: Positive results from **Conduct Diplomacy** or **Sacrificing a Unit**.

Summary: Player moves their diplomacy marker on the Diplomacy Track.

Steps:

-

1. System displays Diplomacy Track.

2. User moves Diplomacy marker and confirms movement.

3. System updates Diplomacy Track.

Alternative: User moves Diplomacy marker to a players hex. That neutral is now allied with that player and the diplomacy marker is removed from play.

29 Sacrifice

Actor: Player

Goal: Sacrifice a character or unit in order to sway a neutral's allegiance.

Precondition: Movement is done during movement phase. The sacrifice is done during the diplomacy phase.

Summary: A player sacrifices a unit or eligible character.

Related: Move Diplomacy Marker

Steps:

1. User moves character, unit, or captured character to a hex adjacent to a character or unit of the neutral.

2. During the diplomacy the system removes the "sacrifice"

3. User moves neutral diplomacy marker as in **Move Diplomacy Marker**

Alternative: User isn't allowed to sacrifice characters because of their religion.

30 Toggle Dragon Blockade Status

Actor: Player

Goal: Change the status of blockades in a dragon tunnel complex.

Precondition: The dragon must be inside or adjacent to an entrance of a Dragon Tunnel Complex.

Summary: The player controlling the dragon chooses to either blockade entrances to a tunnel complex, or to remove a blockade. Either of these replace the dragon's movement for that game-turn.

Steps:

1. User selects a dragon player and which type of blockade change will be happening

2. System asks which hex sides will be effected by the change

3. User selects one or more hex sides.

31 Attempt to Escape

Actor: Player

Precondition: Happens during manna regeneration phase. A Player must have a captured character.

Summary: Captured characters can attempt to escape their grisly fate.

Steps:

-

1. System asks during manna regeneration whether the user wants character to attempt escape.

2. User chooses to attempt escape.

3. System calculates results of escape attempt.

Alternatives:

1. User chooses not to escape

2. Magic-using characters are prompted prior to step 3 whether they wish to use manna to aid their escape.

32 Eliminate Prisoner

Actor: Player

Precondition: Diplomacy interphase. A Player must have a captured character in a capitol.

Summary: Captured characters can be executed.

Steps:

-

1. System asks during diplomacy interphase whether the user wishes to execute captured characters in capitols.

2. User chooses to execute the prisoner.

3. System updates state and maps and informs players of the grisly result.

Alternative: User chooses not to execute (for now).

33 Eliminate Unit from Full Stack

Actor: Player

Precondition: The player ends an inter-phase with an over-sized stack that has to be trimmed.

Summary: The player selects which units to eliminate from a stack that is too full

Related: Select from stack

Steps:

-

1. The game shows the player the units in an oversized stack.

2. The player selects units to eliminate until the stack is small enough.

System and Software Design Description(SSDD):
Incorporating Architectural Views and Detailed Design Criteria
For
Swords and Sorcery(S&S)

Version 1.0

Prepared by:
University of Idaho Computer Science 383 Class, Spring 2014

Prepared for:
Dr. Clinton Jeffery

May 9, 2014

Swords and Sorcery Design
Table of Contents

1   Introduction

This is the System and Software Design Document for the computer adaptation of the Swords and Sorcery board game. This is one of five documents that describe the computer adaptation of the Swords and Sorcery board game. The computer adaptation was developed by the Software Engineering class at the University of Idaho in Spring 2014.

1.1   Document Purpose, Context, and Intended Audience

1.1.1   Document Purpose

The purpose of this document is to describe the system and software design of Swords and Sorcery. This includes diagrams developed to guide design of S&S, component descriptions, view descriptions, and requirements traceability.

1.1.2   Document Context

This document is written as part of a larger document that describes the Swords and Sorcery project developed by the CS383 students at University of Idaho, in Spring 2014. This document only describes the system and software design of the project, which is only a subset of the project.

1.1.3   Intended Audience

This document is intended to be read by Dr. Clinton Jeffery and members of the class, as well as any interested members of the University of Idaho Computer Science department. This document is not intended to be distributed publicly in any way.

1.2   Software Purpose, Context, and Intended Audience

1.2.1   System and Software Purpose

The purpose of the Swords and Sorcery system and software is to provide a computer adaptation of the complex board game of the same name. The system is designed to provide multiplayer functionality over the internet, and to simplify the complex rules of the original Swords and Sorcery.

1.2.2   System and Software Context

The context of this project is, again, restrained to the classroom, as it is an educational project, not intended for distribution. However, the source code for the project, as well as many resources, are available publicly on github.com.

1.2.3   Intended Users of System and Software

The intended users of the Swords and Sorcery system are the developers (students of CS383) and any outgoing, motivated individuals who find the S&S source on www.github.com. Also included in intended users is the class instructor, Dr. Clinton Jeffery.

1.3 Definitions, Acronyms, and Abbreviations

Term	Definition
AD	Architectural Description: "A collection of products to document an architecture." ISO/IEC 42010:2007
Alpha Test	Limited release(s) to selected, outside testers.
Architectural View	"A representation of a whole system from the perspective of a related set of concerns." ISO/IEC 42010:2007
Architecture	"The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." ISO/IEC 42010:2007
Army Unit	An instance of the class ArmyUnit, which is a subclass of MovableUnit.
Beta Test	Limited release(s) to cooperating customers wanting early access to developing systems.
Client	The process the user directly interacts with, containing, among other things, the GUI.
Design Entity	"An element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced." IEEE STD 1016-1998
Design View	"A subset of design entity attribute information that is specifically suited to the needs of a software project activity." IEEE STD 1016-1998
Edge	The edge between two hexes. Edges can include roads, walls, streams, etc. and can effect movement or combat
FX Scene Builder	A tool used for easily creating layouts in FXML to work with javaFX.
FXML	FXML is a declarative XML-based language created by Oracle Corporation for defining the user interface of a JavaFX 2.0 application.
Edge Element	A particular element (such as a road) on a given edge, one edge can contain multiple elements.
GUI	Graphical User Interface - What the user sees and interacts with - also called the HUD.
Hex or Hexagon	A hexagon shaped location on the game or diplomacy map that can contain things like units, edges, or terrain. Or the mathematical hexagon shape.
Map	A logical 2D collection of hexagons based off of the physical S&S game board or diplomacy map
HexID	A unique hex identification string.
HUD	Heads Up Display - What the user sees, with respect to interface - also called the GUI.
IP	Internet Protocol - Typically refers to an IP Address.
JavaFX	A software platform for creating and delivering rich applications that can run on a wide variety of devices.
JSON	JavaScript Object Notation - a lightweight data format based off of the javascript programming language that is easy for both humans and machines to read and write.

Pane	A container for some form of action or information. JavaFX has many, including Anchor Panes, Grid Panes, Box Panes and more.
PlayerID	An integer representing one of the factions described in this scenario.
S&S	Swords and Sorcery
Scene	The fxml file that is to be loaded into a stage for displaying to the screen.
Server	The (single) process that a client connects and sends messages to.
SSDD	System and Software Design Document
SSRS	System and Software Requirements Specification
Stage	This is essentially the container for the application. Load different scenes into the stage to switch which is visible at the time.
System	Ä collection of components organized to accomplish a specific function or set of functions."ISO/IEC 42010:2007
System Stakeholder	Än individual, team, or organization (or classes thereof) with interests in, or concerns, relative to, a system."ISO/IEC 42010:2007
Unit	An instance of a MoveableUnit

1.4   Document References

1. CSDS, *System and Software Requirements Specification Template*, Version 1.0, July 31, 2008, Center for Secure and Dependable Systems, University of Idaho, Moscow, ID, 83844.

2. ISO/IEC/IEEE, *IEEE Std 1471-2000 Systems and software engineering – Recommended practice for architectural description of software intensive systems,* First edition 2007-07-15, International Organization for Standardization and International Electrotechnical Commission, (ISO/IEC), Case postale 56, CH-1211 Geneve 20, Switzerland, and The Institute of Electrical and Electronics Engineers, Inc., (IEEE), 445 Hoes Lane, Piscataway, NJ 08854, USA.

3. IEEE, *IEEE Std 1016-1998 Recommended Practice for Software Design Descriptions*, 1998-09-23, The Institute of Electrical and Electronics Engineers, Inc., (IEEE) 445 Hoes Lane, Piscataway, NJ 08854, USA.

4. ISO/IEC/IEEE, *IEEE Std. 15288-2008 Systems and Software Engineering – System life cycle processes,*Second edition 2008-02-01, International Organization for Standardization and International Electrotechnical Commission, (ISO/IEC), Case postale 56, CH-1211 Geneve 20, Switzerland, and The Institute of Electrical and Electronics Engineers, Inc., (IEEE), 445 Hoes Lane, Piscataway, NJ 08854, USA.

5. ISO/IEC/IEEE, IEEE Std. 12207-2008, *Systems and software engineering – Software life cycle processes*, Second edition 2008-02-01, International Organization for Standardization and International Electrotechnical Commission, (ISO/IEC), Case postale 56, CH-1211 Geneve 20, Switzerland, and The Institute of Electrical and Electronics Engineers, Inc., (IEEE), 445 Hoes Lane, Piscataway, NJ 08854, USA.

1.5   Overview of Document

Section 2   of this document describes the concerns and constraints of the system and software, with respect to environmental constraints, system requirement constraints, and user characteristic constraints. Section 2 also describes stakeholder concerns.

Section 3 of this document describes the System and Software architecture of Swords and Sorcery, from different points of view, namely the user's point of view and the developer's point of view.

Section 4 of this document describes the finer details of the software design, listing software and system components that are crucial to the operation of the Swords and Sorcery game.

Section 5 of this document describes the requirements traceability of the Swords and Sorcery project, highlighting how our original project requirements have been met, modified, and implemented.

1.6 Document Restrictions

This document is for LIMITED USE ONLY to UI CS personnel working on the project.

2 Constraints and Concerns

2.1 Constraints

Swords and Sorcery requires the Java Runtime Environment 8 to run. S&S also requires the JDK 8.0 and Netbeans 8.0 or better to develop, or a proficiency with ANT, which is the directory structure the project uses, as a result of being developed in Netbeans. The game will run on Windows, Mac, and Linux, provided those systems have the mentioned software packages (JRE to play, JDK/Netbeans to develop). To play the game, a network connection is required, as well as the IP Address of the game server. As S&S is a multiplayer game, one client is required for each player. Typically, this should be done using multiple computers, however, multiple clients can be run on the same computer.

2.2 Stakeholder Concerns

There are no financial stakeholders, however, Dr. Clinton Jeffery can be considered a stakeholder, as well as each class member. As a class member, our concerns are providing a quality product, while Dr. Jeffery's concerns may include using good design principles, as this is a Software Engineering course. Other concerns include design requirements such as portability,

3 System and Software Architecture

3.1 Developer's Architectural View

3.1.1 Developer's View Identification

There are multiple developer views within the scope of the S&S project for CS383. The views represent each subteam, and there are three subteams - HUD Team, Rules Team, and Networking Team. The descriptions of each sub-view follow, as well as an overview diagram.

HUD Team View The HUD Team view includes all software design related to the HUD/GUI and handles how the user(s) interact with the S&S game rules and networking.

Rules Team View The Rules Team view includes all software implementations of the S&S rule set, some of which overlaps with other views. Typically, Rules Team is in charge of implementing internal logic and data structures.

Networking Team View The Networking Team view includes all network communication related to the S&S game. This includes the client/server model, the communication protocols, the server setup and more.

3.1.2 Developer's View Representation and Description

Architectural Overview The following class diagram is inaccurate with respect to the project as it currently stands. The inaccuracies are the Chart class, which does not exist, and we discovered is unnecessary, as well as the alliance and player classes, which were never created. The alliance class was never created because we didn't make it that far with the project, and the player class was replaced with flags and variables, as a class was determined to be unneeded. Also, the diagram lacks references and design information relating the classes to the HUD and Networking components of S&S.

Overview Class Diagram Author: Keith Drew
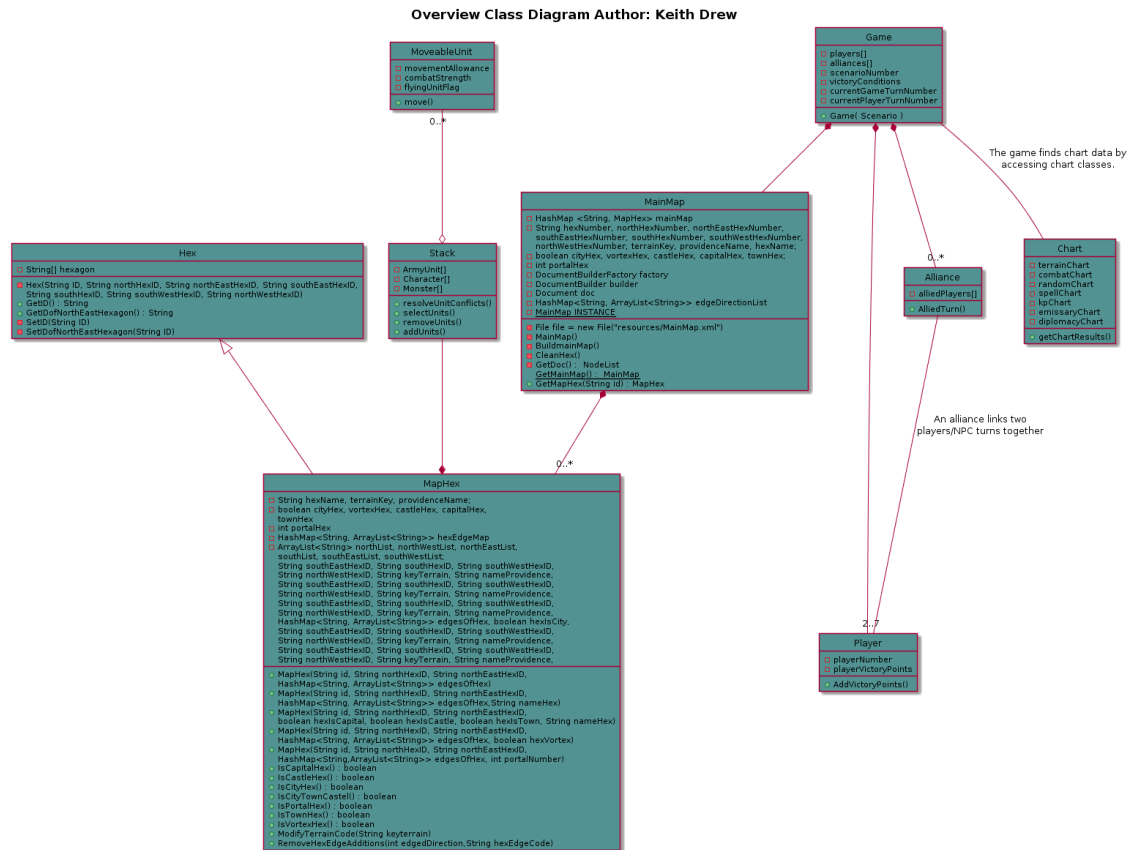
HUD Team View Representation and Description The HUD team view includes rules that affect the HUD (movement, hex painting, terrain, etc.) and the actual HUD itself, containing a view of the map, the minimap, character/unit descriptions, the solar display, the main menu, and hosting interactions between the game instance and the user. The HUD team also designed several data structures include the UnitPool and MainMap, for displaying the units and map in the HUD. Following is a collaboration diagram that shows a rough idea of the design from the HUD Team view.

Collaboration Diagram
HUD Team Overview

Author: David Klingenberg, Keith Drew, Colin Clifford, Jay Drage
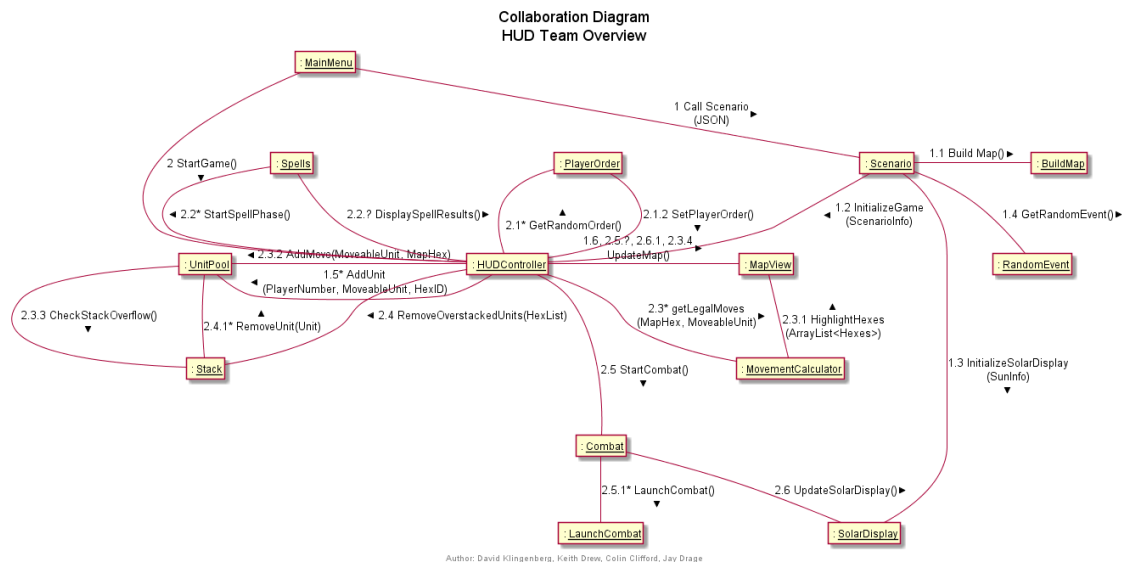
Rules Team View Representation and Description The rules team view covers the implementation of the rules from the actual boardgame, S&S. This includes things like combat, spell casting, charts implementation, unit implementation and other rules related tasks. The design view for the rules team consists of how those rules interact with each other and the other groups (HUD, Networking). Following is their design for MoveableUnits, which many of their components were dependent on.

Unit Class Diagram Author: Ian Westrope
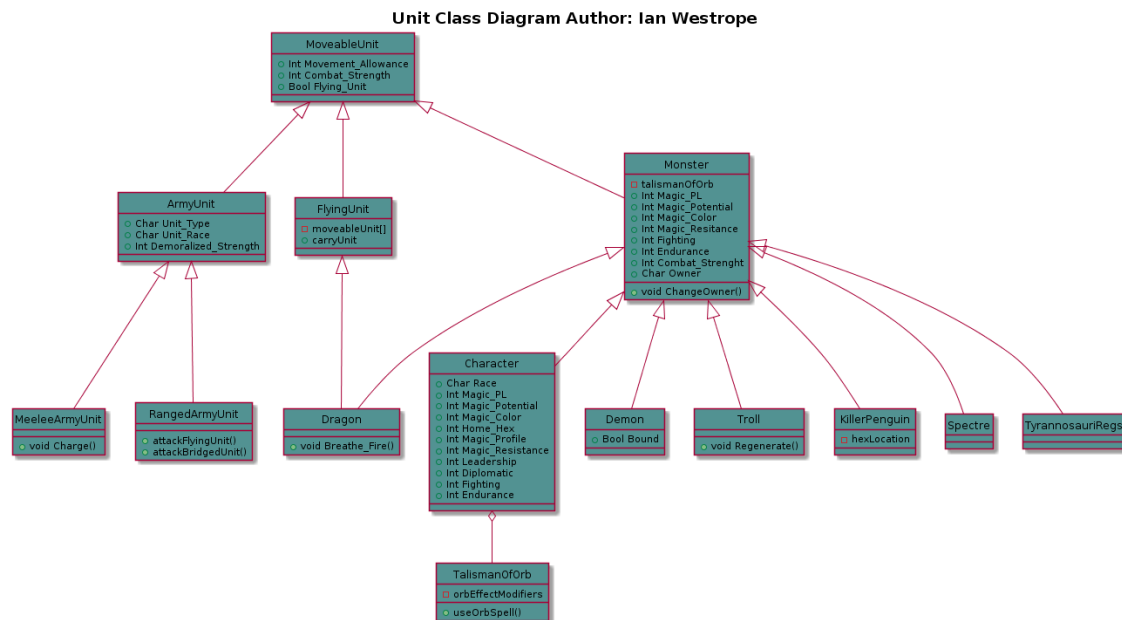
Networking Team View Representation and Description The networking team view covers the client/server implementation and design for S&S. Their overall view of design was a client/server model which hosts communication over the network and message passign for updating the state of a game.

Networking Client-Server Organization
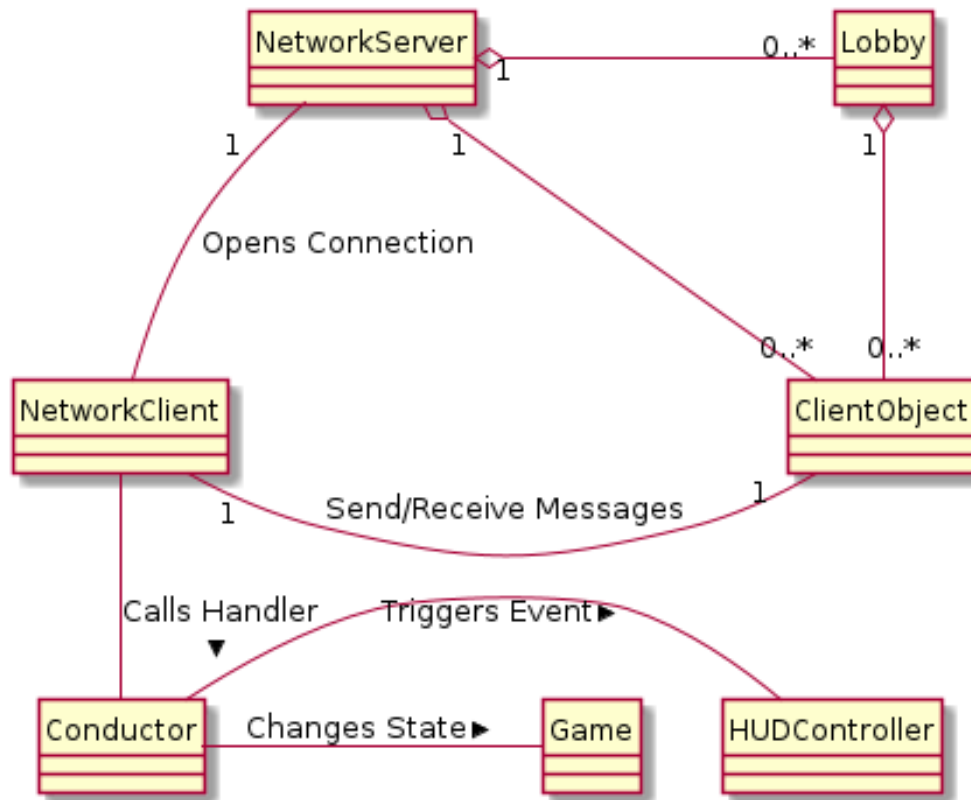
Sending and Receiving Messages In-Game
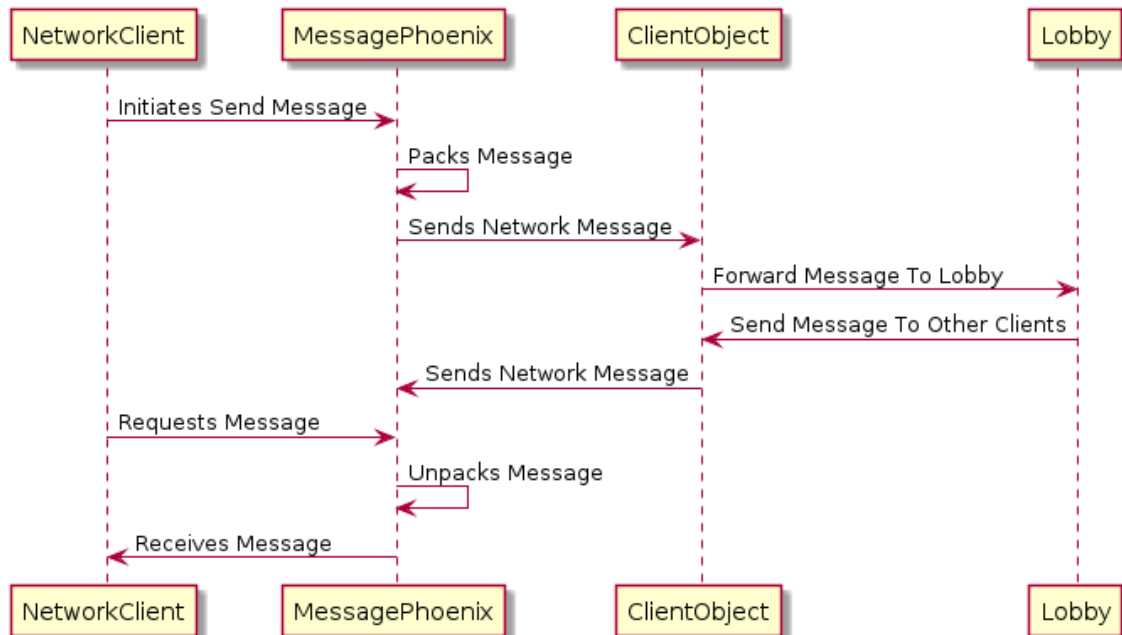
3.2 User's Architectural View

3.2.1 User's View Identification

The User's view consists of their interactions with the S&S GUI. This includes starting the client, joining/starting a game lobby, beginning a game, and interacting with the game. Found here are diagrams representing sequence of play and mouse/key button presses.

3.2.2   User's View Representation and Description

Sequence of Play
by Cameron Simon
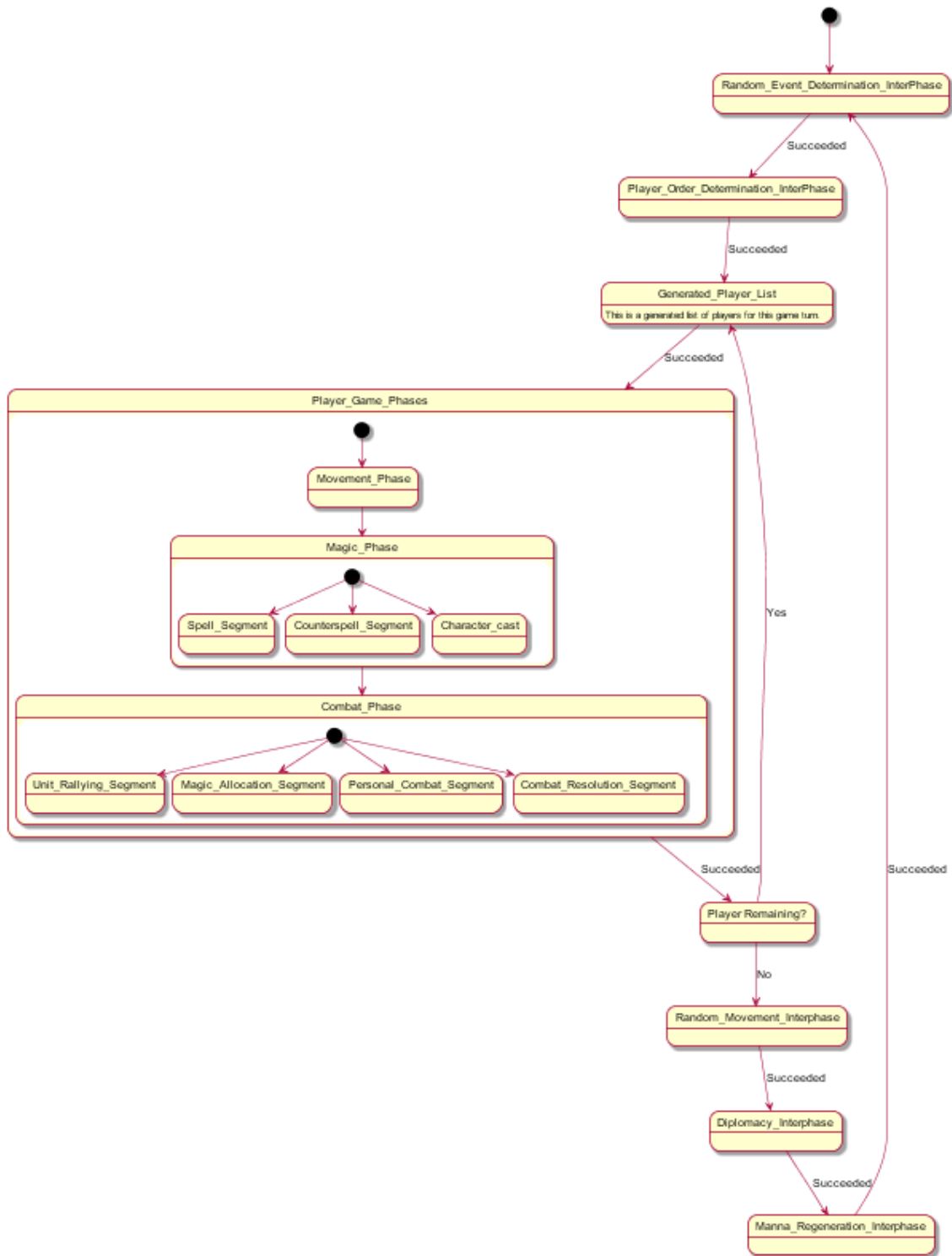
Keyboard and Mouse buttons GUI statechart
Jay Drage

Main Game Map
show main menu popup

select end game

[esc]

[esc]

Main Game Map
mouse in main map panel

[esc]

left mouse

[esc]

Unit Selected
highlight selected unit

after attack

[esc]

[d]

after move

[m]

[esc]

[c]

[esc]

[a]

[esc]

Display Unit Info

Unit Ready to Move
show range of movement

after spell

Unit Ready to Cast
show available spells

Unit Ready to Attack

left mouse target hex

Move Unit

select spell

Cast Spell

left mouse target hex

Complete Attack

4 Software Detailed Design

4.1 Developer's Viewpoint Detailed Software Design

This section describes the developers viewpoint with respect to the software design. Included are the UML diagrams developed for the S&S game, from which we began designing S&S, as well as the state and collaboration diagrams. Many of the diagrams were developed at different points during the semester, reflected by the discrepancies between the SSRS, this document, and the Implementation document.

4.1.1 UML Class Diagrams

Master Class Diagram
Authors: Keith Drew, Ian Westrope

EdgeElement
+ getEdgeType()
+ makeEdgeElement(String attr)
+ makeEdgeElement(HexEdgeType t)

0..6

HexMap
- DocumentBuilderFactory
- DocumentBuilder
- Document
- file
- GetColumns
+ BuildMap()
+ GetHex(String)
+ GetHex(int, int)
+ GetRows()
+ makeHex(Node)
+ LowFirstRow()
+ GetIDFromCoords(int, int)
+ GetCoordsFromID(String)

HexEdge
+ HexEdge(String hex, int dir)
+ put(EdgeElement e)
+ getHexEdgeType t)
+ getEdgeID()
+ contains(HexEdgeType t)

HexEdgeType
- enum HexEdgeType

0..*

Hex
- String[] hexagon
+ Hex(String ID, String northHexID, String northEastHexID, String southEastHexID, String southHexID, String southWestHexID, String northWestHexID)
+ GetID() : String
+ GetIDofNorthEastHexagon() : String
+ SetID(String ID)
+ SetIDofNorthEastHexagon(String ID)

MainMap
- HashMap <String, MapHex> mainMap
- String hexNumber, northHexNumber, northEastHexNumber, southEastHexNumber, southHexNumber, southWestHexNumber, northWestHexNumber, terrainKey, providenceName, hexName;
- boolean cityHex, vortexHex, castleHex, capitalHex, townHex;
- int portalHex
- DocumentBuilderFactory factory
- DocumentBuilder builder
- Document doc
- HashMap<String, ArrayList<String>> edgeDirectionList
- MainMap INSTANCE
- File file = new File("resources/MainMap.xml")
- MainMap()
+ BuildmainMap()
+ CleanHex()
+ GetDoc() : NodeList
+ GetMainMap() : MainMap
+ GetMapHex(String id) : MapHex

DiplomacyMap
- HashMap <String, DiplomacyHex> DiplomacyMap
- String hexNumber, northHexNumber, northEastHexNumber, southEastHexNumber, southHexNumber, southWestHexNumber, northWestHexNumber, specialCode
- DocumentBuilderFactory factory;
- DocumentBuilder builder
- Document doc
- File file
- DiplomacyMap INSTANCE
- DiplomacyMap()
+ BuildDiplomacyMap()
+ GetDiplomacyMap() : DiplomacyMap
+ GetDiplomacyHex(String id) : DiplomacyHex

1..*

MapHex
- String hexName, terrainKey, providenceName;
- boolean cityHex, vortexHex, castleHex, capitalHex, townHex
- int portalHex
- HashMap<String, ArrayList<String>> hexEdgeMap
- ArrayList<String> northList, northWestList, northEastList, southList, southEastList, southWestList;
- String southEastHexID, String southWestHexID, String northWestHexID
- String northWestHexID, String keyTerrain, String nameProvidence,
- String southEastHexID, String southHexID, String southWestHexID,
- String northHexID, String keyTerrain, String nameProvidence,
- String northWestHexID, String keyTerrain, String nameProvidence,
- HashMap<String, ArrayList<String>> edgesOfHex, boolean hexIsCity,
- String southEastHexID, String southHexID, String southWestHexID,
- String southEastHexID, String southHexID, String southWestHexID,
- String northWestHexID, String keyTerrain, String nameProvidence,
- MapHex(String id, String northHexID, String northEastHexID, HashMap<String, ArrayList<String>> edgesOfHex)
- MapHex(String id, String northHexID, String northEastHexID, HashMap<String, ArrayList<String>> edgesOfHex,String nameHex)
- MapHex(String id, String northHexID, String northEastHexID, boolean hexIsCapital, boolean hexIsCastle, boolean hexIsTown, String nameHex)
- MapHex(String id, String northHexID, String northEastHexID, HashMap<String, ArrayList<String>> edgesOfHex, boolean hexVortex)
- MapHex(String id, String northHexID, String northEastHexID, HashMap<String,ArrayList<String>> edgesOfHex, int portalNumber)
- IsCapitalHex() : boolean
- IsCastleHex() : boolean
- IsCityHex() : boolean
- IsCityTownCastel() : boolean
- IsPortalHex() : boolean
- IsTownHex() : boolean
- IsVortexHex() : boolean
- ModifyTerrainCode(String keyterrain)
- RemoveHexEdgeAdditions(int edgedDirection,String hexEdgeCode)

DiplomacyHex
- boolean playerHex, neturalHex
- String southEastHexID, String northHexID, String southWestHexID,
- String southEastHexID, String southHexID, String southWestHexID,
+ public DiplomacyHex(String id, String northHexID, String northEastHexID, String northWestHexID, String specialHexCode)
+ DiplomacyHex(String id, String northHexID, String northEastHexID, String northWestHexID, String specialHexCode)
+ SetPlayerHex()
+ SetNeturalHex()
+ GetIsPlayerHex() : boolean
+ GetIsNeturalHex() : boolean

TerrainType
+ getMovementCost(MoveableUnit unit)
+ getCombatMultiplier(MoveableUnit unit)
+ getCombatEffect(MoveableUnit unit)
+ makeTerrainType(String code)

ImprovedTerrainType
+ ImprovedTerrainType()
+ improvedTerrainType(MapHex h)
+ getMovementCost(MoveableUnit unit)
+ getCombatMultiplier(MoveableUnit unit)
+ getCombatEffect(MoveableUnit unit)
+ getMovementOverride(MoveableUnit unit)
+ makeImprovement(String code, MapHex h)

Unit Class Diagram Author: Ian Westrope

MoveableUnit
o Int Movement_Allowance
o Int Combat_Strength
o Bool Flying_Unit

ArmyUnit
o Char Unit_Type
o Char Unit_Race
o Int Demoralized_Strength

FlyingUnit
- moveableUnit[]
o carryUnit

Monster
- talismanOfOrb
o Int Magic_PL
o Int Magic_Potential
o Int Magic_Color
o Int Magic_Resistance
o Int Fighting
o Int Endurance
o Int Combat_Strenght
o Char Owner
+ void ChangeOwner()

MeeleeArmyUnit
+ void Charge()

RangedArmyUnit
+ attackFlyingUnit()
+ attackBridgedUnit()

Dragon
+ void Breathe_Fire()

Character
o Char Race
o Int Magic_PL
o Int Magic_Potential
o Int Magic_Color
o Int Home_Hex
o Int Magic_Profile
o Int Magic_Resistance
o Int Leadership
o Int Diplomatic
o Int Fighting
o Int Endurance

Demon
o Bool Bound

Troll
+ void Regenerate()

KillerPenguin
- hexLocation

Spectre

TyrannosauriRegs

TalismanOfOrb
- orbEffectModifiers
+ useOrbSpell()

Data Structures Class Diagram Author: Keith Drew

GameCombatResultsChart
- combatResultsLookupArray[]
- getCombatResults()

Chart
- terrainChart
- combatChart
- randomChart
- spellChart
- kpChart
- emissaryChart
- diplomacyChart

GameRandomEventChart
- randomEventLookupArray[]
- getRandomEvent()

GameSpellChart
- spellLookupArray[]
- getSpellInfo()

GameKillerPenguinEntryChart
- kpEntryLookupArray[]
- kpCount
- getKPEntry()
- getKPCount()

GameEmissaryResultsChart
- emissaryLookupArray[]
- getEmissaryResults()

StellarConfiguration
- yellowSunPosition : int
- redSunPosition : char
- blueSunPosition : char
- redSunAscendent : boolean
- redSunDescendent : boolean
- StellarConfiguration(positionOfRedSun : char)
- AdvanceSuns() : void
- CheckMinorSunsPosition(minorSunAscendentPositionOne : char, minorSunAscendentPositionTwo : char, minorSunAscendentPositionThree : char) : void

Author: David Klingenberg

GameTerrainChart
- terrainLookupArray[]
- getTerrainDefenseBonus()
- getTerrainMoveCost()
- isPassableEdge()

RandomEvent
- currentEventKey : int
- currentEventLength : int
- currentEventDescription : int
- RandomEvent(cEKey : int, cELength : int, cEDescription : int) : void

Author: David Klingenberg

These use the suns

GameDiplomaticGroupChart
- diplomacyGroupLookupArray[]
- getDiplomacyResults()

The DipMap does not live here
but is accessed here.

DiplomacyMap
- HashMap <String, DiplomacyHex> DiplomacyMap
- String hexNumber, northHexNumber, northEastHexNumber, southEastHexNumber, southHexNumber, southWestHexNumber, northWestHexNumber, specialCode
- DocumentBuilderFactory factory;
- DocumentBuilder builder
- Document doc
- File file
- DiplomacyMap INSTANCE
- DiplomacyMap()
- BuildDiplomacyMap()
- GetDiplomacyMap() : DiplomacyMap
- GetDiplomacyHex(String id) : DiplomacyHex

1..*

DiplomacyHex
- boolean playerHex, neturalHex
- String southEastHexID, String southHexID, String southWestHexID, String southEastHexID, String southHexID, String southWestHexID,
- public DiplomacyHex(String id, String northHexID, String northEastHexID, String northWestHexID, String specialHexCode)
- DiplomacyHex(String id, String northHexID, String northEastHexID, String northWestHexID, String specialHexCode)
- SetPlayerHex()
- SetNeturalHex()
- GetIsPlayerHex() : boolean
- GetIsNeturalHex() : boolean

4.1.2   UML Collaboration Diagrams

Collaboration Diagram
Move Use Case

: MovementCalculator

: MapHex

1.2 getValidMoves(unitID,
hex,moveAllowance,
validMoves)

◄ 2.1 clearHighlights()

1.1 MapHex.getUnits()

1* SelectUnit
(hexID, hex) ►

: MapView

1.3 highlight
(validMoves, color)

: HUDController

1.4 DisplayStack
(unitPane, SelectedStack) ◄

2 MoveUnit
(hexID, hex) ◄

2.3 DisplayStack
(unitPane, SelectedStack)

2.2 addMove(selectedUnit, hex.GetID()
▼

: UnitPool

Author: David Klingenberg

4.1.3   UML State Charts

State Chart for Captured Character by Ian Westrope

Terrain Magic / Random Effects

States and transitions (Terrain Magic / Random Effects):
- Forested Devastation
- Normal
- Devastated
- Forested
- Melted
- Crossable
- Frozen
- Frozen Bridged
- Bridged
- Blasted

Transitions: Wizard Wheel, Forest, Windstorm, Firestorm, FireStorm, End of Drought, Drought, River Crossing, Wizard Wheel / Timeout / Dismissed, Ersatz Winter, Wizard Wheel / End of Winter, Wizard Wheel, Building, Disintegration, End of Winter, Ersatz Winter, Wizard Wheel / Disintegration

Solar Cycle
Author: David Klingenberg

Start Red Sun at [A,C, .., K]

Start Red Sun at [B,D, .., L]

Red Sun [A,C, .., K]

Suns in Balance
Count <= 6

Depending on start code, it is possible to start in any state and at any point in the count [1<=c<=5] or [1<=c<=6].

Count > 6

Red Sun Descendent
Blue Sun Ascendant
Count <= 5

Count > 5   Count > 5

Suns in Balance
Count <= 6

Count > 6

Red Sun Ascendant
Blue Sun Descendent
Count <= 5

Red Sun [B,D, .., L]

Suns in Balance
Count <= 5

Depending on start code, it is possible to start in any state and at any point in the count [1<=c<=5] or [1<=c<=6].

Count > 5

Red Sun Descendent
Blue Sun Ascendant
Count <= 6

Count > 6   Count > 6

Suns in Balance
Count <= 5

Count > 5

Red Sun Ascendant
Blue Sun Descendent
Count <= 6

Scenario Ends   Scenario Ends

4.2   Component Dictionary

Name	Type-Range	Purpose	Dependencies	Subordinates
AddMove	UnitPool Method	Move a unit to a new hex	Two array lists: hexList, unitMove	None
AddUnit	UnitPool Method	Add a unit to the sorted tree map	Tree map: pool	None
Army Combat Result Table	Static Method	Determine results of combat lookup	Two ArrayLists: Attackers, Defenders	None
Army Unit	Class	Unit SubClass	Moveable Unit	All individual unit types
Characters	Class	Unit SubClass	Moveable Unit	None
Clear	UnitPool Method	Cleans up the UnitPool for testing purposes.	The entire data structure	None
ClearOverStack	UnitPool Method	Clears an over-stacked array.	Sorted map: over-StackMap	None
ClientObject	Class	Represents an open connection to a client from the server.	Tag, Flag, MessagePhoenix	
Conductor	Class	Contains public handler methods for incoming network methods.	Tag, Flag	
Diplomacy	fxml file	Fxml used by Diplomacy Controller	Diplomacy Controller	
Diplomacy Controller	Controller Class	Controls the Diplomacy map scene	Depends on HUD Controller, MapView, and Scenario.	
End Movement Phase	UnitPool Method	Prepares the UnitPool for the next move phase.	Sorted map: unitMove	None
Flag	Enum Class	An enumerate constant class providing identifiers for each concrete type of network message.		
Game	java	Starts the application and sets up stage.		
Get All Player Units	UnitPool Method	Gets all player units.	Tree map: pool	None

GetInstance	UnitPool Method	Gets a Singleton instance of UnitPool.	None	None
GetOverStack	UnitPool Method	Gets hexes in violation of over-stack rule.	Tree map: over-StackMap	None
Get Player Specific Units	UnitPool Method	Get all units by type owned by a player.	Sort map: pool	None
getRand-SafeHex	Static Method	Given a list of provinces, return a pseudo-random hex ID from the map that is in those provinces and not water.	MainMap, MapHex	None
Get Safe Teleport	UnitPool Method	Determine if a unit can safely teleport.	Array list: safeTeleport	None
Get Teleport Destination	UnitPool Method	Get a unit destination portal.	Sorted map: portalNum	None
GetUnit	UnitPool Method	Retrieves a units.	Sorted map: pool	None
Get Unit Hex Move	UnitPool Method	Retrieves all hexes a unit has moved through.	Array list: unitMove	None
Get Unit In Hex	UnitPool Method	Retrieves all units in a hex.	Array list: hexList	None
Hexagon Classes	Model	Represents a hexagon	Hex Edge Classes, Terrain Classes, UnitPool	Hexagon
Hex Edge Classes	Model	Classes to collect and represent elements on hexagon edges	Hex Classes	Hexagon Classes
HexStack	Class	Ensure compliance with the games stacked rules.	UnitPool	None
Hex Stacked	Class	Implements rules for stacks.	UnitPool	None
Hexagon Map Classes	Model	Classes to represent a "map" of hexagons, either diplomacy or game.	Hexagon Classes	MapView, Hexagon Classes
hud	fxml	Layout for the game Hud	HUD Controller	None
HUD Controller	javafx controller	Handles actions and events triggered in hud scene	hud.fxml	None
Launch Combat	Static Method	Implement Combat Phase	Moveable Unit, Army Combat Results Table, HUD-Controller, Movement Calculator	None
Lobby	Class	A server-side object that manages a grouping of client connection.	Tag, Flag, Mes-sagePhoenix	

mainMenu	fxml	Layout for the main menu	Main Menu Controller.java	None
Main Menu Controller	JavaFX Controller	Handles actions and events triggered in mainMenu	Scene	mainMenu.fxml
Hex Rendering	View	Renders the game map and things on it	MapView, Hexagon Classes, Hex Edge Classes	
MapView	View	A GUI widget to display the results of the Hex Rendering code	Hexagon Classes, Hex Rendering	JavaFX GUI
MessagePhoenix	Utility Class	Used to pack, unpack, send and receive messages over the network	Tag, Flag	
Moveable Unit	Class	Unit SuperClass	None	ArmyUnit, Character, Monster
Movement Calculator	Static Class	Determine Legal Moves	UnitPool, MainMap	Retreat/Move
NetworkClient	Class	Manages network connection for client.	Tag, Flag, MessagePhoenix	Conductor
NetworkServer	Class and Process	The independent server process that clients connect to.	Tag, Flag, MessagePhoenix	ClientObject
Over Stack Warning	Static Method	Notifies player of an over-stack condition.	UnitPool	HUD Controller
PopulatePool	Static Method	Fill the unit pool with appropriate units/characters in their specified locations.	Provinces, Characters, Units, object creator, character maker, getRandSafeHex()	None
Remove Over Stack	Method	Removes units from an over-stacked hex.	UnitPool	HUD Controller
RemoveUnit	UnitPool Method	Removes a single unit.	Sorted map: pool	None
Retreat	Static Method	Implement Retreat after Combat	Launch Combat, Movement Calculator	None

Scenario	Singleton Class	Store data about game scenario, for components, fill unit pool with units and characters appropriately.	JSONArray, JSONObject, JSON simple parser, Provinces, Characters, Units, object creator, character maker	populatePool()
SetSafe Teleport	UnitPool Method	Sets the units that can safely teleport.	Array list: safeTeleport	None
Solar Display	Class	Tracks solar position, updates HUD image	Scenario/GameRules	Spell Casting, HUD Controller
Spell Cast	Static Class	Perform Spell effects on given characters/units	None	
Tag	Enum Class	An enumerated constant class providing identifiers for each general type of network message.		
Teleport	UnitPool Method	Teleports a unit to a new portal.	Sort of map: pool	None
UndoMove	UnitPool Method	Undo a unit's previous move.	Two array lists: hexList, unitMove	None
UnitPool	Class	Track and manipulate all units in the game.	None	MovableUnit, hexStack

4.3   Component Detailed Design

4.3.1   Detailed Design for Component: Army Unit

Purpose   This class contains the data of the Army Units in the game and extends the Movable Unit class. This class contains new data like the strength of a unit and whether or not the units are conjured or demoralized. If a unit is conjured then there are special member variables that contain values that are associated with conjured units. If a unit is demoralized then the strength of the unit is different so a demoralized strength variable was added to the class. The strength and demoralized strength variables are used during the combat phase of a users turn and is used to determine the outcome of combat.

Input   The only Inputs to this class are the ones needed to fill the member variables of this class.

Output   This class by itself has no output produced other than the getter functions in the class.

Process   Output is obtained by calling the getter functions.

Design Constraints and Performance Requirements In order for this class to pre-form correctly all of the needed variables need to be filled out.

4.3.2 Detailed Design for Component: Army Combat Results Table

Purpose The purpose of this method is a lookup for the results of Combat.

Input Input needs to be two Array Lists: one is called attackers and one defenders. Also the hex object that the defenders are positioned on is needed. The attackers array list is comprised of all of the attacking Army Units in the combat and the defenders are all of the defending Army Units in the combat. The defenders hex is needed in order to calculate the defence multipliers that the defending units get from the terrain values of the hex object.

Output The output of this function is a simple 2 value array corresponding the result of the combat. The first value is what's required of the attackers and the second value is what's required of the defenders. A -1 means the units were killed, a 0 means that there was no result of the combat and any positive number represents the number of hexes a unit has to retreat.

Process The function adds the total strengths of both the attackers and defenders. Then applies the terrain multiplier to the defenders total strength. Finally the ratio of attackers over defenders plus a random dice roll determines the outcome of the combat.

Design Constraints and Performance Requirements One design constraint of the table was that in the game the ratio's have to be reduced to the smallest possible fraction in favour of the defending units. To work around this and index was made to match the determined ratio to the correct look up value on the table. Also the units strength and race is required to be filled out in order for this function to work properly.

4.3.3 Detailed Design for Component: ClientObject

Purpose ClientObject is used by the server. ClientObject is a class which represents a client who has connected to the server. ClientObject is not something that runs on the client machine. NetworkServer creates an instance of ClientObject for each new connection to the server. ClientObject is responsible for the socket to the client. The main activity of a ClientObject instance is to listen for incoming messages from the client represented by the ClientObject, which it accomplishes by an independent thread, and to send messages to the client through the network socket.

Input ClientObject receives messages from the associated client.

Output NetworkServer and other ClientObjects are allowed to send message to the associated client through ClientObject.

Process ClientObject's listener thread reads and process messages from the connected client. Other threads request the writer

Design Constraints and Performance Requirements

4.3.4 Detailed Design for Component: Conductor

Purpose   The Conductor class is used by the client. When NetworkClient detects an incoming message that alters the state of the game (such as a unit being moved), it calls a method inside of the Conductor class. The purpose of putting handler code in the Conductor class rather than inside of NetworkClient was to seperate the code in NetworkClient that deal with internal networking objects (like sockets) from the code that deals with other game object (like unitPool).

Input   Each message in the Conductor class is invoked with parameters received via an incoming network message.

Output   The methods in the Conductor class may respond to an incoming message through any public interface. For example, the Conductor class may alter UnitPool in response to a network message.

Process   An incoming message is received inside of NetworkClient. NetworkClient determines the type of message, and forwards the message to Conductor is appropriate. Conductor checks the tag of the message, to determine the message type, and calls the appropriate code for the message type.

Design Constraints and Performance Requirements   Conductor was designed to let other team members work conveniently with networking code, without having to stare at networking internal details.

4.3.5 Detailed Design for Component: DiplomacyController

Purpose   Controls the Diplomacy map scene

Input   N/A

Output   Shows diplomacy map

Process   This class sets a swingnode content to that of a MapView instance of the diplomacy map and also has a button that returns the scene to that of the HUD.

Design Constraints and Performance Requirements   This class depends on the HUDController class which is where it is called from.

4.3.6 Detailed Design for Component: Diplomacy.fxml

Purpose   The fxml used for displaying the diplomacy map scene.

Input   N/A

Output   The diplomacy map scene.

Process   It is read in and then displayed by selecting Diplomacy from the HUD

Design Constraints and Performance Requirements   N/A

4.3.7   Detailed Design for Component: MessagePhoenix

Purpose   MessagePhoenix contains the methods to create, send, and receive messages over the network. This functionality is used by both the client and server. MessagePhoenix is intended to be called indirectly by client code through NetworkClient, (as well as by the Server).

Input   MessagePhoenix requires a reference to an input or output object stream associated with a network socket. The utility methods in MessagePhoenix also accept any number of Objects (using a variable length parameter list), which will be packaged and sent over the network. The first two objects of a message must be a Flag and Tag, which identify the message.

Output   MessagePhoenix can return the NetworkPacket received from a network connection.

Process   Receiving a message initiates a blocking read from the network socket. Sending a message writes to the network socket immediately.

Design Constraints and Performance Requirements   MessagePhoenix needs to accomodate a variety of message types.

4.3.8   Detailed Design for Component: NetworkClient

Purpose   NetworkClient is used by the client. NetworkClient provides an interface that other client-side components can use to interact with the network. NetworkClient creates the connection to the server, and sends and receives messages over the network.

Input   NetworkClient listens for incoming messages from the network. Some messages impact the internal state of NetworkClient, and other messages are forwarded to Conductor.

Output   NetworkClient provides a public interface for sending message over the network.

Process   To send a message, NetworkClient uses MessagePhoenix along with the network socket. On receiving a message, NetworkClient may respond internally, or forward the message to Conductor is the message concerns non-networking parts of the code (like unit movement).

Design Constraints and Performance Requirements   NetworkClient must be able to receive network message asynchronously.

4.3.9   Detailed Design for Component: NetworkServer

Purpose   NetworkServer is the main process that runs on the server machine. It waits for incoming connection requests. It creates a ClientObject for each connected client, and manages the group of connected clients through their ClientObject representations.

Input   NetworkServer receives connection requests from client processes.

Output   NetworkServer creates new threaded ClientObject instances for each connection.

Process   When a connection is opened, the ClientObject instances is created (initiating an exchange of information, like user names, between the client and server), and stored in a list of connections.

Design Constraints and Performance Requirements NetworkServer must be efficient enough to handle the expected number of connections. For our pupuoses, the demands on the NetworkServer are fairly minimal, and few performance issues have arisen.

4.3.10 Detailed Design for Component: Tag

Purpose The Tag class contains "message tags". A message tag is the second object in a network packet. The tag identifies the concrete type of message. For example, the "SEND HANDLE" tag identifies a message as containing the handle (the username) of the new connection. By examining the Tag of a message, we can correctly interpret the other contents of the message.

Input The Tag class receives no input.

Output The Tag class does not produce output.

Process None.

Design Constraints and Performance Requirements None.

4.3.11 Detailed Design for Component: Flag

Purpose The Flag class contains "message flags". A message flag is the first object in a network packet. The flag identifies the general type of a message. For example, there is a "REQUEST" and "RESPONSE" flag. The motivation for this is because many interactions with the server follow a REQUEST, ACCEPT or DENY format. For example, you might "REQUEST" "SEND HANDLE" in one message, and expect a "RESPONSE" "SEND HANDLE".

Input None.

Output None.

Process None.

Design Constraints and Performance Requirements None.

4.3.12 Detailed Design for Component: Lobby

Purpose The Lobby class is used by the server. Lobby represents a grouping of client connections (ClientObjects, which live on the server), and is used to manage a game instance. Lobby can remember things like the current turn.

Input A Lobby can receive messages from the NetworkServer, or forward messages from the connected ClientObjects.

Output A Lobby can forward messages received from one ClientObject to all clients in the Lobby.

Process Clients are added to or removed from (by client request) a particular lobby.

Design Constraints and Performance Requirements None.

4.3.13 Detailed Design for Component: LaunchCombat

Purpose The LaunchCombat is a public static Java class, which allows players to see combat details, and then they can decide to enforce the Combat between units or not.

Input Two ArrayList <MoveableUnit> attackers and defenders, and the object from MapView so it can highlight those hexes for showing more combat information.

Output Confirmation for players to decide enforce combat, if yes, shows the combat result and give options of retreat or eliminate certain units.

Process The LaunchCombat first takes input Moveable units, this connect with the HUD-controller class: left mouse click to get selected_stack, and right mouse click to get target_stack. When a player in the combat phase, they should pick both units then press ′A′ to launch the combat. When clicked A button, first it shows dialogs to ask the attacker player if they want to add friendly units which surround the defender units into the combat, then it will pop out units detail include Attacks' strength, Defenders' strength, Defenders' Terrain type, and Defenders' strength after Terrain Type bonus with a confirmation dialog. The certain player should click yes button for showing combat result, or click no for doing nothing. If the player clicks yes button, the Combat result will shows as a notification on the right bottom conner, with this result both attacker player and defender player may meet three situations: Nothing Change, Retreat, Elimination. If the combat result returns 0, then the certain player doesn't need to do any reactions from this combat, if the result is a negative number, the certain player should eliminate numbers(since result is a integer number) of units from the units list; if the result is a positive number, then the player should decide to eliminate the unit OR retreat the unit.

Design Constraints and Performance Requirements One Design Constraint is that the LaunchCombat only can be used while the Combat Phase, and both of Attacker Units and Defender Units should be selected.

4.3.14 Detailed Design for Component: Retreat

Purpose After the combat, players should decide to retreat the units if the result requires.

Input ArrayList<ArmyUnit>, combat result

Output It will highlight those hexes which the certain units can retreat to with red color, and right click the mouse button for retreating the certain units to certain hex.

Process First the result came from the LaunchCombat, so the player can get it's result value(negative, positive integer, or zero), if the result is a positive number then the player should decide to retreat the units or not, if not, then they should eliminate the unit, if yes, execute the Retreat function. The retreat function will send necessary input for the method getRetreatMove in the class Movement Calculator, so it can get which hexes that the certain units can move to, and it will highlight the available hexes with red color.

Design Constraints and Performance Requirements Retreat function will only be called while the combat result sent, in the other words, it needs to know the attackers list, defenders' list, and the combat result. For each units will be sent a message to ask for retreating if necessary.

4.3.15 Detailed Design for Component: Spell Cast

Purpose The purpose of this class is to perform spells during the spell cast phase of the game. These spells can have a variety of affects like destroying units, moving units, demoralizing units, along with many more.

Input The character object and user input for things like unit to be cast on, number of targets, or manna to be transferred.

Output The effects to units/characters. This includes things like demoralization, graphical walls, or movement of units on map.

Process This class takes in the necessary information for the spell being cast. Determines what the effects will be based on user input and character magic potential, then displays the effect on the map.

Design Constraints and Performance Requirements In order to perform a spell cast all information for that spell must be known. This includes things like limits, range of spell, distance to target, and character manna potential.

4.3.16 Detailed Design for Component: Characters

Purpose This class contains the data of a Character in the game and extends the Movable Unit class. This class has variables unique to a character: magic level, magic potential, current manna, magic colour, and leadership. The magic level determines the hight level of spell that a character can cast. The magic potential is the maximum amount of manna that a character can have. The magic colour of a character determines when a character's spells have the most effect. Leadership is the influence that a character has in determining the result of army combat. Characters have the special ability to cast spells which uses the magic and manna values to determine the amount and effectiveness of their spells.

Input The only Inputs to this class are the ones needed to fill the member variables of this class.

Output This class by itself has no output produced other than the getter functions in the class.

Process Output is obtained by calling the getter functions.

Design Constraints and Performance Requirements In order for this class to pre-form correctly all of the member variables need to be filled out.

4.3.17 Detailed Design for Component: Solar Display

Purpose The purpose of the solar display class is to maintain the progress of the solar chart, as defined in the rules of S&S. The solar display class also updates the HUDController with the proper information, ie state, to display for the solar chart.

Input The starting locations of the blue and red suns, read from the game scenario being loaded, are the only input to the solar display class.

Output   The output of the solar display class is the information the HUDController needs to update the solar display image. The solar display class returns the image that needs to be displayed in the solar display section of the hud, as well as the state of the suns.

Process   The solar display class performs a rotation by incrementing the positions of each sun, then determines whether the state of the suns change. The states that can be returned are for the blue and red suns, and are dependent upon the yellow sun. The possible states for the blue and red suns are equilibrium, ascension, and descension. The last task the solar display performs is to update the HUD accordingly.

Design Constraints and Performance Requirements   The only design constraints/requirements for the solar display are that the information passed to the HUD is accurate with respect to the S&S rules of gameplay. The solar positioning must also be correctly loaded from the scenario information.

4.3.18   Detailed Design for Component: Movable Unit

Purpose   The purpose of this class is to have a common class of all moving units that the movement functions can access. This class is a super class of all units that can undergo a movement process. This allows for common data to be accessible by the appropriate functions. This common superclass also allows for the ability to store the data of all units in a single-typed data structure. This class contains member variables for movement allocation of a unit, the working movement allocation of a unit or the amount of movement left in a game turn, the race of the unit, the type of subclass that is inheriting from this class(such as armyUnit or Character), and the unique ID of the unit. The movement allocation of a unit is the amount of movement points allowed at the beginning of a turn then the working movement takes over. The working movement is used in order to keep track of how much a unit has move in a single turn. This allows for a user to partially move a unit in their turn then return to that unit and finish moving it later in the same turn. The race is used in many different calculations for units such as the movement cost per hex of a unit in a particular terrain. The subclass variable is used for typecasting the moveable unit back to the proper subclass in order for the subclass based operations to be executed. Finially the unique ID is used for network based communications to identify the unit that is being acted on, as well as determining if two units are friendly/enemy units, based on the owner field of the ID.

Input   Input needed to make this class function properly are values to fill the member variables of this class.

Output   The only output of this class is by getters of the member variables of the class.

Process   Call the getter functions.

Design Constraints and Performance Requirements   One constraint of this class was the necessity to be casted back into the appropriate subclass this was solved by creating the unitType(subclass Type) variable. Also in order for this class to preform right with other classes and functions all of the variables need to be set.

4.3.19   Detailed Design for Component: Movement Calculator

Purpose   The movement calculator is a static java class that handles most forms of movement.

Input   The movement calculator takes two inputs to generate a list of moves: the unit moving, and the hex object they are beginning from. To calculate a retreat, the movement calculator takes as input the unit retreating, the hex they are retreating from, and the number of hexes they are required to retreat.

Output   The movement calculator produces two main outputs: A hashmap of moves that a unit can reach (within the rules of movement specified by the board game) during a given movement phase, paired with their remaining movement cost after moving to a key hex in the hashmap, or an arraylist of moves that a unit can move to while retreating, during the combat phase.

Process   The movement calculator uses recursion to examine the neighbors of the provided hex location. From each neighbor, it evaluates their neighbors, and so on. In both cases (movement/retreat) the recursion is terminated by reaching a lower bound (0) on the limiting value for their movement. For a moving unit, this is their given movement allowance per turn. For a retreating unit, this is the number generated from the combat results table that indicates how many hexes a unit must retreat. For each step of recursion, decisions are made within control flow that are designed to model the rules of the original S&S board game. These factors include, but are not limited to, hex terrain types, hex edge types, geographical obstacles, and enemy occupation.

Design Constraints and Performance Requirements   The design was constrained by two factors - code complexity and time. By designing the movement calculator to use recursion, the complexity of the component was greatly reduced. However, due to the many factors involved in movement, the design is still complex. Also, the moves available to a unit need to be calculated quickly. However, recursion is not very fast. Thankfully, Colin Clifford added some optimization code to the calculator, which has greatly increased performance with respect to time.

4.3.20   Detailed Design for Component: Unit Pool

Purpose   Maintain all units and their positions in the game.

Input   The inputs to this class are units, teleport information, and hex ID's.

Output   The outputs of this class are units and hex ID's.

Process   Methods are in place to control the creation, destruction, and movement of units.

Design Constraints and Performance Requirements   All manipulation of units must occur in this class.

4.3.21   Detailed Design for Component: Hex Stack

Purpose   To enforce the board game rules for stacking units.

Input   Unit IDs and units.

Output   Boolean value indicating if the hex is within the max stack limits.

Process Ensures that the number of units in any given hex at the end of the movement phase are in compliance with stack rules.

4.3.22 Detailed Design for Component: Add Move

Purpose Update the location of a unit

Input Unit and destination hex ID.

Output None

Process Remove the unit from its originating hex and places it in its destination hex.

Design Constraints and Performance Requirements All movement must use this method.

4.3.23 Detailed Design for Component: Add Move Stack

Purpose Use with network update process at the end of the movement phase.

Input Origin hex ID and destination hex ID

Output None

Process All units in an origin hex are moved to the corresponding destination hex.

Design Constraints and Performance Requirements Must only be used in the network update.

4.3.24 Detailed Design for Component: Add Unit

Purpose To add a new unit to the unit pool during scenario creation, replacement, and reinforcement phases.

Input Player ID, unit, and starting location.

Output None

Process The player ID, the class name of the unit, and a unique number are combined together to produce a unique ID for the unit and then it's location is initialized.

Design Constraints and Performance Requirements All units must be created with this function.

4.3.25 Detailed Design for Component: Clear

Purpose It is only for the purposes of testing. Ensures that the unit pool is completely devoid of any units.

Input None

Output None

Process Clears all the data structures.

4.3.26   Detailed Design for Component: Clear Over Stack

Purpose Once All Hexes Are Forced into compliance with the stack rule it clears the stack data structure.

Input None

Output None

Process Calls the Clear Method of the Data Structure.

4.3.27   Detailed Design for Component: End Movement Phase

Purpose Call all housekeeping methods at the end of the movement phase. Including but not limited to stacked methods.

Input None

Output None

Process Call methods to ensure each hex complies with stack rules. Reset appropriate data structures.

Design Constraints and Performance Requirements Ensure that all end of phase movement rules are enforced.

4.3.28   Detailed Design for Component: Get All Player Units

Purpose Get a complete list of all player units for purposes of reinforcements and replacements.

Input Player ID.

Output Tree map of units.

Process Retrieve all units in the pool that belong to the current player.

4.3.29   Detailed Design for Component: Get Instance

Purpose To retrieve the unit pool Singleton.

Input None

Output Unit Pool

Process Retrieves the unit pool Singleton.

Design Constraints and Performance Requirements   The unit pool has to be a Singleton in order to avoid units being out of sync.

4.3.30   Detailed Design for Component: Get Over Stack

Purpose   Retrieve all units violating the unit stacking rules.

Input   None

Output   An array list of units.

Process   At the end of the current movement phase and once all stacks are brought into compliance, the over stacked array is reset.

4.3.31   Detailed Design for Component: Get Player Specific Units

Purpose   To find all units of a specific type that belong to one player.

Input   Player ID and units class name.

Output   An array list of units.

Process   Sorts through the unit pool and retrieves all units of a particular type by player ID.

4.3.32   Detailed Design for Component: Get Safe Teleport

Purpose   Retrieve the list of all units that can safely teleport. This list was generated during the spell phase.

Input   Unit ID.

Output   A Boolean value.

Process   Checks if the unit ID is contained inside of the safe teleport list.

Design Constraints and Performance Requirements   All units attempting to teleport must be evaluated against this flag.

4.3.33   Detailed Design for Component: getUnit

Purpose   Retrieve an instance of a unit from the unit pool.

Input   Unit ID.

Output   Instance of a MoveableUnit.

Process   The unit ID is used to retrieve a instance of a movable unit from the unit pool.

Design Constraints and Performance Requirements   any reference to a instance of a movable unit must come from this method.

4.3.34 Detailed Design for Component: getUnitHexMove

Purpose   Retrieve every movement a unit has made in the current movement phase.

Input   Unit ID

Output   Unit

Process   Looks up the unit's ID and returns hexIDs for each hex the unit has stopped in.

Design Constraints and Performance Requirements   Can only be used during the players current movement phase.

4.3.35 Detailed Design for Component: getUnitsInHex

Purpose   Retrieve all units currently occupying a given hex. In effect it retrieves the stack of units.

Input   Hex ID

Output   A list of unit IDs.

Process   Searches the hex list array for any units it contains.

4.3.36 Detailed Design for Component: removeUnit

Purpose   Eliminate any unit that has been destroyed, from the unit pool.

Input   Unit

Output   None

Process   Removes unit from the sorted tree map.

4.3.37 Detailed Design for Component: setSafeTeleport

Purpose   Sets the flag for all units in a stack, with a casting character, to allow a nondestructive teleport.

Input   An array list of unit IDs

Output   None

Process   Sets a flag for each unit in a stack to true.

Design Constraints and Performance Requirements   Only used during the players current spell phase.

4.3.38 Detailed Design for Component: setTeleportDestination

Purpose   To set the portal number the units will travel to during the spell phase.

Input An array list of unit IDs

Output None

Process During the spell phase each unit in the casters hex will have their destination portal sent.

Design Constraints and Performance Requirements Must only be used during the current players spell phase.

4.3.39 Detailed Design for Component: teleport

Purpose When a unit enters a portal hex it is either moved to a random portal, a predetermined portal, destroyed, or nothing happens. The process is controlled by any flags that may have been set during the spell phase.

Input Unit

Output Boolean value

Process If the safety flag is set the unit is either randomly moved to a new portal or left in place. If the portal number is indicated the new unit will move to that specified portal. If neither teleport flag is set the unit will be teleported to a random portal which includes the possibility of destruction.

Design Constraints and Performance Requirements Only used during the current players movement phase.

4.3.40 Detailed Design for Component: undoMove

Purpose Step back to the previous hex a unit stopped in.

Input Unit ID

Output The previous hex ID.

Process Moved to the previous index of a discrete unit in the unit move data structure.

Design Constraints and Performance Requirements Can only be used during the players current movement phase.

4.3.41 Detailed Design for Component: hexStack

Purpose Implements the games stack rules.

Input The inputs for this class include hexes and units.

Output Boolean values, information indicating stack compliance, and GUI elements used to bring components into compliance with the rules.

Process Checks occupied hexes for compliance with the stack rule. Eliminate excessive units for hexes that are out of compliance with the rules.

Design Constraints and Performance Requirements Only used during the current players movement phase.

4.3.42   Detailed Design for Component: overStackWarning

Purpose Warn the player if they have a stack that is in violation of the rules.

Input Array list of unit IDs

Output Boolean value

Process Counts the number of units in a hex and returns false if there are too many units in a hex.

Design Constraints and Performance Requirements Characters are excluded from the count as they are not army units. Only used during the current players movement phase.

4.3.43   Detailed Design for Component: removeOverStack

Purpose To bring a stack into compliance with the rules.

Input Sorted map of units identified by hex ID

Output None

Process Displays a GUI showing all the units in all the stacks. Allows you to select the units to be eliminated. Then eliminates the selected units.

Design Constraints and Performance Requirements Characters are excluded from the count as they are not army units. Only used during the current players movement phase.

4.3.44   Detailed Design for Component: mainMenu.fxml

Purpose This file contains all the layout panes, buttons and labels for the Main Menu of the game as well as their sizes and positions.

Input This requires no input.

Output Visually displays a layout for the Main Menu.

Process This is loaded into a scene and displayed when placed in the stage.

Design Constraints and Performance Requirements This layout was designed for compatibility with virtually any screen resolution. But does start to cut things off when it reaches the dimensions of the "Swords & Sorcery" label.

4.3.45 Detailed Design for Component: hud.fxml

Purpose   This file contains all the layout panes, buttons and labels for the in game HUD as well as their sizes and positions.

Input   This requires no input.

Output   Visually displays a layout for the HUD.

Process   This is loaded into a scene and displayed when placed in the stage by a call from the MainMenuController.java.

Design Constraints and Performance Requirements   This layout was designed for compatibility with virtually any screen resolution. But does start to cut things off when It reaches the dimensions of the game information panel on the right side of the screen as these dimensions do not scale.

4.3.46 Detailed Design for Component: MainMenuController.java

Purpose   This manages the controls of the buttons in the mainMenu.fxml layout. It is used for loading the game/scenario and quitting the application completely.

Input   Action calls from buttons on the mainMenu layout.

Output   There is no output for this.

Process   This is designated as a target in the corresponding fxml file to handle all of its actions. It's functions are called by actions and events specified in the layout.

Design Constraints and Performance Requirements   Every functions can only be called via actions from the layout and therefore sometimes you must make a function that is called by your action and only calls an external function.

4.3.47 Detailed Design for Component: HUDController.java

Purpose   This manages the controls of the mapview, selected units, target units, chat/message boxes, and the menu bar in the hud.fxml layout. It is used for playing the game and displaying necessary information about the game.

Input   Action calls from buttons, the main map, keyboard, and chat server.

Output   Outputs messages to the game server.

Process   This is designated as a target in the corresponding fxml file to handle all of its actions. Its functions are called by actions and events specified in the layout.

Design Constraints and Performance Requirements   Every functions can only be called via actions from the layout and therefore sometimes you must make a function that is called by your action and only calls an external function. We also implemented an initialize function that loads in and sets up the game map and other initial functionality.

4.3.48 Detailed Design for Component: Game.java

Purpose   This starts the application.

Input   No input for this.

Output   No output for this.

Process   This loads the scenes with their appropriate fxml files then loads the stage with the appropriate scene.

Design Constraints and Performance Requirements   A JavaFX main file is static. This is a mentionable note because it effects how you handle switching scenes and calling information from the scene in other java files.

4.3.49 Detailed Design for Component: Hexagon Classes

Purpose   To collect data pertaining to a hexagon

Input   Hexagons are constructed from sections of XML data. The original design called for things like terrain or random effects.

Output   Hexagons have various members that can be used to check the state of the hex.

Process   Hexagons are constructed with the help of the Hex Map classes. This process also sets up Hex Edges to be shared between Hexes. From there the state of a Hexagon can be changed through various functions. The state of a hex can also be indirectly changed by moving a unit in the UnitPool as the MapHex class reflects the state of the UnitPool with respect to units in a hex.

Design Constraints and Performance Requirements   It was important that hexes could be queried for what hexes or edges neighbor them. Hex IDs correspond to the numbering scheme from the game map.

4.3.50 Detailed Design for Component: Hex Edges

Purpose   To collect data pertaining to an edge between two hexagons

Input   Hex Edges are constructed from sections of XML data. The original design called for things like force-walls.

Output   Hex Edges have various components. As well as neighboring hexes.

Process   Hex Edges are constructed from parts of XML data. Each Hex edge could contain 0 or more Edge Components. Components are organized in a standard Java collections class.

Design Constraints and Performance Requirements   It was important that when two neighboring hexes point to the edge between them that it's the same edge object regardless of what side you looked from.

4.3.51 Detailed Design for Component: Map Classes

Purpose To act as a collection to store Hexagons in a logical 2D plane.

Input Maps are constructed with .XML data. They use this .XML data to construct hexagons, and then store the hexagons in a standard Java collection.

Output Hex IDs can be converted to actual hexagons.

Process Two types of Map classes. MainMap and DiplomacyMap. They store their respective types of hexes. Apart from that pretty simple code to look up hexes in the hashmap when one is requested.

4.3.52 Detailed Design for Component: Hex Rendering

Purpose To render anything visible on the map.

Input MapView calls the Hex Rendering code, and passes it a hex to draw, a hex to highlight, or an edge to draw. It also passes in a Graphics2D surface to draw on.

Output The Hex Rendering code renders onto the Graphics2D surface.

Process It is accomplished with swings 2D API. Images are loaded from the resources folder to help draw units or terrain. Edges are drawn using solid straight lines. Highlighting and unit background colors are done with transparency.

Design Constraints and Performance Requirements Use swing to render stuff.

4.3.53 Detailed Design for Component: Map View

Purpose To be a GUI frontend widgit to the Hex Rendering.

Input The GUI framework calls various event driven functions according to GUI stuff.

Output Map View works with the Hex rendering code to paint onto it's Graphics surface. This surface is displayed by the GUI system. Converting GUI coordiantes to hex coordinates is also supported.

Process Lots of the calls from the GUI framework are trivial. But one important method is paintComponent, which is passed a Graphics object to paint on when painting should happen. It does this by using a for loop over all the hexes it should render, and passing that to the hex rendering code. Apart from this there are methods to turn GUI coordinates into Hex IDs or edge coordinates. This is done with math similar to the following: http://gamedev.stackexchange.com/a/20762

Design Constraints and Performance Requirements Same as Hex Rendering. An extra constraint after initial implementation was to work with JavaFX.

4.3.54 Detailed Design for Component: Scenario

Purpose This class is tasked with reading in data from the scenario configuration JSON files and temporarily storing it within the Java program. Most of this data is then accessed and processed by other components of the project. This data includes anything one may want to know about a scenario including its name, number of players, blue sun initial position, the armies and nations participating, as well as their provinces, units, and characters and additional information on neutrals and diplomacy. In addition to storing information, the Scenario class must also populate the unit pool with characters and units of the scenario in their appropriate provinces.

Input The Scenario is told what file to load the data from by a call to its static method Initialize(). From this file, it reads the scenario information.

Output Scenario stores information and will add units and characters to the unit pool at their appropriate hex addresses.

Process The Scenario is told what file to load the data from by a call to its static method Initialize(). From this file, it reads the scenario information, looping through complex data structures where necessary. Once populatePool() is also called from the outside, it loops through stored data about units, characters, and their provinces in order to add these objects to the unit pool in appropriate hexes.

Design Constraints and Performance Requirements While no hard performance requirements were specified, it was implied that reading the scenario file and populating the unit pool should not require an unreasonable amount of time such that the user will become impatient.

4.3.55 Detailed Design for Component: populatePool()

Purpose The pool populator traverses through the data acquired about each player nation and neutral so that it may fill the unit pool with the coorect units and characters in the appropriate provinces.

Input The data already contained in the Scenario class.

Output Several new additions to the unit pool that reflect the units and characters posessed by neutrals and player nations in the Scenario.

Process For each player army in the Scenario, the unit pool traverses through its nations, keeping track of the controlling player. For each of these nations (which have a distince nation and race), the units and characters lists are traversed. Appropriate units are added to that player's branch of the unit pool in random hexes scattered throughout the provinces controlled by the nation. Then neutral nations are traversed in a similar manner.

Design Constraints and Performance Requirements Must be accurate and relatively timely.

4.3.56 Detailed Design for Component: getRandSafeHex()

Purpose Return a pseudo-random hex ID that is not water terrain from a list of provinces.

Input   A list of provinces and the data contained in MainMap.

Output   A random hex ID that does not correspond to a water type and is in the listed provinces.

Process   First a pseudo-random province from the list is chosen. Then a pseudo-random hex is chosen from the list of hexes in that province. While this hex is not water, a new random one is chosen. The final choice is returned.

Design Constraints and Performance Requirements   Must be accurate and relatively timely.

4.4   Data Dictionary

Name	Type/Range	Defined By...	Referenced By...	Modified By...
allowance Cache	HashMap	Movement Calculator	Movement Calculator	Movement Calculator
retreat Allowance Cache	HashMap	Movement Calculator	Movement Calculator	Movement Calculator
HexMap	class/HashMap	HexMap.java	MainMap, DiplomacyMap	HUDController, ...
UnitPool	Class, SortedMap	UnitPool.java	Movement Calculator, Networking, MainMap, Spell, Combat, Stack	HUDController, Networking, Spell, Combat, Stack
Pool	SortedMap	UnitPool.java	UnitPool	UnitPool
HexList	SortedMap	UnitPool.java	UnitPool	UnitPool
UnitMove	SortedMap	UnitPool.java	UnitPool	UnitPool
PortalNum	SortedMap	UnitPool.java	UnitPool	UnitPool
SafeTeleport	ArrayList	UnitPool.java	UnitPool	UnitPool
OverStackMap	SortedMap	UnitPool.java	UnitPool	UnitPool
INSTANCE	UnitPool	UnitPool.java	UnitPool	UnitPool
Stack	Stack	UnitPool.java	UnitPool	UnitPool
Options	Final object[2] ,[0]Yes, [1]No	UnitPool.java	UnitPool	None
PortNum	Int	UnitPool.java	UnitPool	UnitPool
HexMap's HexID map	HashMap	HexMap.java	Hexagon Map Classes	Hexagon Map Classes
Hex Edge Elements	TreeMap	HexMap.java	Hexagon Map Classes	Hexagon Map Classes
HexStack	Class	HexStack.java	UnitPool	UnitPool
UnitCount	SortedMap	HexStack.java	UnitPool	UnitPool
UnitRemoveList	ArayList	HexStack.java	UnitPool	UnitPool
PopupScene	Scene	HexStack.java	UnitPool	UnitPool

Count	Integer	HexStack.java	UnitPool	UnitPool
Scenario	singleton class, collection of ints, strings, bools, String Lists, Maps, and Maps of Maps.	Scenario.java	Game, Solar-Config	Game

5   Requirements Traceability

5.1   Components

5.1.1   Movement

Requirements Description   Our requirement for movement was that a unit would be selected from the GUI and the GUI would highlight all available moves for the given unit. The player could then select the desired location for movement and the unit would move there.

Implementation Description   Our implementation of movement uses recursion to generate a list of available moves that are highlighted in the GUI. The moves are then displayed as highlighted hexes. When the controlling player then right-clicks the desired hex (within the highlighted set), the unit moves to the indicated hex.

Differences   There is no difference between our requirement for movement and our implementation of movement.

5.1.2   Unit HashMap

Requirements Description   This was left completely out of the design.

Implementation Description   Implemented a way to track all the units in the game.

Differences   It was added as a design modification during one of our sprints. The rules of the game required tracking of units, manipulation of persistent units, a way to create, and destroy those units. The class name is UnitPool.

5.1.3   Stack Class

Requirements Description   Movable units aggregated into stacks, which then formed composites of map hexes. The class was originally named Stack.

Implementation Description   The stacked class now aggregates into the unit pool.

Differences   It is a completely redesigned aggregation. The unit pool class ended up tracking all of the units instead of the individual Map Hex as originally designed. It was only logical to redesign the class diagram and associated aggregations. The name of the class had to be changed to HexStack as the "Stack" class is already exists in the Java libraries.

5.1.4 DiplomacyController

Requirements Description   Allow player to move Diplomacy Marker.

Implementation Description   We implemented a diplomacy map view, but it has no functionality towards displaying current game status.

Differences   This difference stems from time and priority constraints. The diplomacy map was deemed low priority compared to combat, magic, movement, and scenarios.

5.1.5 Combat

Requirements Description   The Combat should allow player to pick more than two units on the map, check their relations(Friendly units or not), then shows their combat information, and a confirmation for the combat. After the combat it should give choice to retreat or eliminate certain units.

Implementation Description   There has several part in the combat phase need to take care about. We designed two adapters with HUDController class and the Retreat and Elimination Class. So the LaunchCombat class will handle everything about combat, it pass necessary variables into the Movement Calculator and the Combat Result Table, after get the return values, it will shows the commands for players with GUI.

Differences   There is no much difference between the requirement and our implementation, only the Retreat part we decided to make it a randomly select an available hex for moving.

5.1.6 MoveableUnit

Requirements Description   We required units, but didn't specify how they should be designed. However, we designed them.

Implementation Description   Every unit we have implemented, that can move on the map, inherits from MoveableUnit.

Differences   We didn't have anything, and we built units.

5.2 Hex Rendering

Requirements Description   Render spells, special hexes, Army Units, Characters, Combat Indications, Special Effects, Edges, Terrain, and anything else required by the army game.

Implementation Description   Render most Army Units, Terrain, and Edges.

Differences   Rendering wasn't specified in the design, but there were lots of design elements that would require specific rendering support.
   Not all of which got implemented.
   Rendering code went pretty smoothly, so this was mostly just from lack of time to add everything in the face of more pressing concerns.

5.2.1 MapView

Requirements Description Swing GUI widget to use Hex Rendering code to render Hexes in a game or diplomacy map.

Implementation Description Swing GUI widget inside a SwingNode JavaFX class. Renders hexes in a game or diplomacy map.

Differences The JavaFX integration was unforseen, but apart from that there aren't many major differences. There was some talk of stuff like being able to pan with the mouse, but this didn't happen due to time constraints.

One notable problem is JavaFX / Swing threading constraints being broken, it's suspect that this is leading to this component throwing exceptions on some computers.

5.3 Hexagon, Edge, and Map Classes

Requirements Description Hexagons have neighbors and tessalate onto a map. Hexagons can be modified by persistent spell or random event effects. Hexagons know what stacks they contain. Hexagons have a terrain type and edges.

Implementation Description MainMap or DiplomacyMap class loads itself from XML file.

This constructs a bunch of hexes, and the edges between them.

Map Hexes have an association with UnitPool to know what units are in them, but are otherwise pretty static.

Map Hexes have terrain, edges, and a combat and movement cost.

Differences In the end Hexagons ended up being pretty static.

To fully support the design they'd have to react to things like spells or random events. But this did not get implemented.

5.4 Hex Rendering

Requirements Description Render spells, special hexes, Army Units, Characters, Combat Indications, Special Effects, Edges, Terrain, and anything else required by the army game.

Implementation Description Render most Army Units, Terrain, and Edges.

Differences Rendering wasn't specified in the design, but there were lots of design elements that would require specific rendering support.

Not all of which got implemented.

Rendering code went pretty smoothly, so this was mostly just from lack of time to add everything in the face of more pressing concerns.

5.4.1 MapView

Requirements Description Swing GUI widget to use Hex Rendering code to render Hexes in a game or diplomacy map.

Implementation Description Swing GUI widget inside a SwingNode JavaFX class. Renders hexes in a game or diplomacy map.

Differences   The JavaFX integration was unforseen, but apart from that there aren't many major differences. There was some talk of stuff like being able to pan with the mouse, but this didn't happen due to time constraints.

One notable problem is JavaFX / Swing threading constraints being broken, it's suspect that this is leading to this component throwing exceptions on some computers.

5.5   Hexagon, Edge, and Map Classes

Requirements Description   Hexagons have neighbors and tessalate onto a map. Hexagons can be modified by persistent spell or random event effects. Hexagons know what stacks they contain. Hexagons have a terrain type and edges.

Implementation Description   MainMap or DiplomacyMap class loads itself from XML file.

This constructs a bunch of hexes, and the edges between them.

Map Hexes have an association with UnitPool to know what units are in them, but are otherwise pretty static.

Map Hexes have terrain, edges, and a combat and movement cost.

Differences   In the end Hexagons ended up being pretty static.

To fully support the design they'd have to react to things like spells or random events. But this did not get implemented.

5.6   Traceability Analysis

5.6.1   Attack Units

Combat between units has been implemented well, but different from the original design in that we should have added friendly units which surround defenders in the combat. And after the Attack Units we should decide the certain player to retreat or eliminate units, that's the feature what we didn't design before.

5.6.2   Retreat

For the retreat part, we finally designed to let units retreat to a randomly available hex.

5.6.3   Choose Leader

Instead of choose the leader, in this project we used List to implement stack units. So far our work doesn't really need to pick who is the leader in the game.

5.6.4   View Character/Unit Statistics

There has several different kinds of situation need to show unit's statistics. We implement different styles for showing details for different situation. For example, while a player picks a units, it will shows the information on the main frame, but when the player want to enforce combat, there will pop out another windows for Combat Details.

6   Appendix A

Swords and Sorcery Implementation Description
University of Idaho, CS 383, Spring 2014
May 9, 2014

1 Introduction and Document Description

This is a description of the Swords and Sorcery implementation, both in general and the implementation of major code modules.

This document particularly describes what was not covered in the design documentation, as well as places where implementation differs from design.

Sections are:

- Implemetation Overview

- Deployment, System Requirements, and Libraries

- Building

- Major Code Modules

2 Implementation Overview

Swords and Sorcery is programmed in Java 8 (after switching from Java 7), and is cross platform across Windows, Mac and Linux.

It uses a 2D top-down view and graphics based on the physical game by SPI, has a GUI programmed in JavaFX and swing, and TCP/IP networking support.

Unfortunately the implementation remains incomplete, so while some basic functionality exists, users will find it extremely difficult to face each other in battle over the internet.

What works or partially works (to the best of our knowledge) includes:

1. Starting a new game (partial)

2. Selecting a Scenario

3. End Inter-Phase

4. Move

5. Teleport

6. Select Unit From Stack

7. Attack

8. Advancing Units (partial)

9. Retreating (partial

10. View Unit Statistics

11. Spellcasting (partial)

12. Starting a client

13. Starting a server

14. Connecting to a server and communicating back and forth

15. Tagged Chat: Lobby-wide and Server-wide

16. Show online users by lobby/server

17. Lobbies: show, create, join, leave

18. Viewing Army Units, Terrain, or basic Hex Edges on the Main Map

19. Viewing a blank Diplomacy Map

20. Seeing the solar display or game phases change when ending a turn

What does not work (to the best of my knowledge) includes:

1. Join Game

2. Set Up

3. Resume Game

4. Save Game

5. Select Ally Candidate

6. Carry

7. Choose Leader

8. Spend Combat Manna

9. Rally

10. Manna Transfer

11. Diplomacy or Emissaries or Prisoners

12. Sacrifice

13. Dragon Blockades

14. Game State Synchronization between clients

15. Special Hex Rendering

16. Character or Monster Rendering

17. MiniMap or Diplomacy Map

18. Random Events

19. User to user chat

Apparently we also have a nasty memory leak, which prevents the program from running for too long. Impressive.

3   Running, Deployment, System Requirements, and Libraries

Swords and Sorcery requires the Java 8 JRE installed to run. This must be installed by the user independantly before trying to launch the game.

There is no installer package. The program takes the form of two .jar files (one for the client, and one for the server), along with a library folder and a resources folder.

In Windows the user should be able to launch the program by double clicking one of these .jar files.

In other supported operating systems the user should be able to launch the program with Java from the command line, with one of the following commands to launch the client or server respectively:

1. java -jar client.jar

2. java -jar server.jar

The program must be launched from the command line in this manner if the user wishes to view debug output.

The only required libraries are

- json-simple 1.1.1
- controlsfx 8.0.5

.jar distributions of these two libraries are included.

Swords and Sorcery isn't the fastest Java code on the block, but should be runnable by most contemporary systems running Windows, Mac, or Linux.

The folder for a swordsorc distribution should contain the following:

1. lib - containing library jars

2. resources

3. client.jar

4. server.jar

4 Building

Swords and Sorcery is hosted on github. If git is installed then the source can be downloaded with the following command:

git clone https://github.com/cjeffery/sworsorc.git

Swords and Sorcery can be compiled from inside Netbeans 8 by selecting either the ¡default config¿ or the Game build targets to build the client. The server can be built by selecting the Server target.

Other build targets exist for debugging or legacy reasons and should be ignored.

Alternatively Swords and Sorcery can be built from the command line with ant, using a build script similar to the following:

ant -f build.xml -q -Dconfig=Game jar ant -f build.xml -q -Dconfig=Server jar

to build the client and server respectively.

Be aware that however the project is built, both client and server will build a file called sworsorc.jar

If both client and server have to exist side by side, then the .jar files will have to be renamed to avoid overwriting eachother.

5 Major Code Modules

Swords and Sorcery has several major code modules. Amongst them the following have interesting enough implementations to bear discussing:

- The JavaFX HUD

- The Map Rendering and the Map View Code

- The Networking System

- The Map File Format

- The Movement Calculator

- Various Gameplay Classes

6   The JavaFX HUD

JavaFX was chosen over swing for the GUI because it's newer and more active than swing. And because it supports scaling natively (as opposed to needing to use kludges to scale stuff in swing).

JavaFX organizes user interfaces into Stages and Scenes. A stage displays a scene. Very theatrical.

JavaFX uses .fxml files to describe interfaces. These can be thought of as similar to HTML.

mainMenu.fxml describes the layout for the Main Options menu which is the first menu that is seen on program launch.

Likewise hud.fxml describes the layout of the actual game screen.

JavaFX also has classes called Controller Classes, that determine the behavior of programs that use these layouts.

In our project MainMenuController.java controls the Main Menu, and is in charge of, for instance, calling a function that actually starts the game when the user hits the "Start Game" button.

HUDController likewise controlls the Main Game screen and has several notable features:

- Load the game map into it's panel

- Update the Sun Position Display at the end of every turn

- Select units when the map is clicked on. And display their stats

- Send messages from the chat box to the server, and vice versa

7 JavaFX MapView interaction

The GUI was switched to JavaFX partway through the project.

This proved a challenge for the Map View implementation, as there are unique challenges using swing widgets in JavaFX based code.

Passable support was fortunately very doable using the SwingNode class.

But one unresolved implementation issue was threading concerns.

Any interaction between swing and JavaFX has to be done on the appropriate thread. But there wasn't enough time to resolve these threading concerns.

The code seems to work OK, but this bug could conceivably lead to weird or inexplicable behavior on certain systems or edge cases.

8 The Map Rendering Code and the Map View

None of the Map Rendering code or Map View code was designed. It was just pulled out of thin air, using the time-honored alchemical process of spontaneous generation.

Fortunately this worked pretty well, as Java's libraries dictate a lot of the design up front, and I (Colin) have had some experience in the past with Java swing programming.

8.1 The Map Rendering Code

There are a few questionable design decisions, and a bit of duplicate code, but overall it worked pretty well and the implementation didn't end up inherently broken.

The Map Rendering code is done with Java's swing framework. All rendering is done onto standard swing Graphics2D objects that are passed in from the Map View.

It consists of two main classes that work in similar ways:

1. UnitPainter

2. HexPainter

Both of these classes are similar, and if they were ever refactored it might make sense to have them both inherit from the same class.

Both classes have methods that accept a Graphics2D object from Java's 2D API to render on to.

Both classes load a variety of PNG images from disk to assist in rendering. Most of the terrain images were drawn by Joe Higley. The unit images were drawn by Colin Clifford.

HexPainter is in charge of rendering Hexagons and edges, and passes off Unit rendering to UnitPainter.

Unit Rendering involves drawing a partially transparant unit icon, and (in the case of Army Units) a status string.

Due to time constraints only Army Units are mostly supported. With characters being stubbed in and looking weird.

Edges are rendering using simple straight colored lines using swing. There's a bit of hexagon math to prevent two elements of the same edge from obscuring eachother, and to draw roads / trails from the center of the hexagon to the edge.

8.2   The Map View Code

Most of the Map View code is in a single class called (appropriately enough) MapView.

MapView is a Swing widget, but the GUI is JavaFX based. So the GUI wraps MapView inside a SwingNode object.

MapView does two main things:

1. Implement various swing widget methods (most notably PaintComponent)
2. Contain methods for necessary math for determining hexagon tiling

MapView implements scrollable, so it can be added to a JScrollPane to support scrolling.

The PaintComponent method is what interacts with the Map Rendering code.

The hexagon tiling math is interesting

8.2.1   Hexagon Math

Hexagon math was cobbled together from various sources on the internet and frenzied scribbles on note-paper.

It's honestly pretty hairy, so I tried to include helpful comments in MapView.

I will not fully describe all the hexagon math here, but I'll share the general principles.

Hexagons are oriented such that the top and bottom are horizontal lines, as per the physical Swords and Sorcery map.

The width of one hexagon is 64px. But as columns fit together smoothly the distance between a column and it's neighbors is 3/4ths that.

The height of a hexagon is the radius (32px) times the square root of 3.

But the key to hexagon tiling was this stackoverflow post: http://gamedev.stackexchange.com/a/20762

Which is to say, Hexagon picking involves converting between hexagon shaped and rectangle shaped tilings. Pretty cool.

With this I was able to tile hexagons graphically, and convert mouse coordinates to hexagons or hexagon edges.

8.3   Interaction Between MapView and Map Rendering

The Map Rendering code has three entry main entry points:

1. Draw a hexagon

2. Draw a hexagon edge

3. Highlight a hexagon

Each of these take a Graphics2D object that is assumed to be pre-translated so that the Rendering Code only has to draw off of the origin.

The first two cases used to be merged, but it was important to avoid drawing a given edge twice, and to avoid an edge being obscured by a neighboring hexagon.

So MapView takes the following passes:

1. Requests the Map Rendering code to draw all the hexagons

2. Requests the Map Rendering code to highlight any hexagons that need highlighting

3. Requests the Map Rendering code to draw all the edges

This "layered" approach gives a pretty smooth look, where nothing draws on top of stuff it shouldn't too much.

9 The Networking System

The goal was to implement network communication of state changes, basic lobby funcationality, and basic chat.

Since we did not integrate our design with the rest of the game at the beginning, there was no logging subsystem setup to track any and all changes to game state. Therefore, our implementation utlimatly involved trying to get individual parts working by hardcoding messages in the requisite areas of code.

9.1 Networking Model

The Networking model used is the infamous client/server model established by Quake. The basic model is thus: clients who want to play together connect to a server, and the server facilitates the connection between them, and sometimes takes on the processing chores of clients as well.

In our implementation of the model, there are "thick" clients that handle almost all of the processing chores, with the server's primary job being to forward messages between clients and facilitate network functionality such as Lobbies, Chat, and Polling. This is similiar to the peer to peer model used by RTS games, with thick clients processing messages recieved, but just with a server inbetween.

Of course, this opens more opportunies for cheating than I want to think of, and an incredible amount of vulnerabilities. If this game were to hypothetically go mass-market, there would need to be a major overhaul before release, for the sake of user security(not to mention online game quality)

9.2 Messages

Network Messages consist of the following:

- FLAG: message category (ex: GAME or CHAT)

- TAG: specific type of message (ex: LOBBY combined with a CHAT flag for Lobby chat)

- Sender: String specifying who sent the message, either the handle of a user, global server, or a specific sub-system

- Data: a List of POJOs (Plain Old Java Objects) which encapsulates any messages being sent. This is a semi-rough hack to allow for (almost) any class to be sent over the network, which we needed with units, spells, and the like

This was better than sending strings, since creating the command parsing and execution framework, where strings send over the network execute commands on the client recieving to change the state, would've possibly required a complete redesign of the core code.

It was not, however, anywhere near decent. The major issues we ran into were parsing the list of data, which led to null pointer exceptions and strange output, and the fact that any object sent needed to be Serializable. JavaFX, unfortunatly, is not, and this led to exceptions and terminated connections whenever a class with a JavaFX object was attemped to be sent over the network.

The message format did serve its purpose, however, and we were able to get some limited funcationality using it.

9.3   Threaded Connections

From the beginning, the network design included theads. Initially, it was simply a sending thread and a recieving thread. However, that expanded into the use of sending/recieving, and command processing threads client side, and sending/recieving threads dedicated to each unique client connection on the server side.

Each thread handles one "half" of the connection, so sending thread would write to the socket, and the recieving thread would read from the socket. Modern ethernet networks support two-way communication using switches, so the old 10Base2 Bus model is certainly no longer viable. Token ring was briefly considored, but would cause more syncronization issues than it would solve, again considoring the technology available.

Queues were implemented to pass messages in and out of threads, basically acting as a "buffer", so if there were slow-downs(common occurance in networks), the threads wouldn't be causing data to be dumped.

The threads themselves didn't cause an abnormal amount of issues, and served their job well. If this were to be put on, again, the "hypothetical market", there would need to be changes to addres performance. Since (far as chris understand it) the threads are a client library, they aren't fully utilizing concurrence available by using OS level threads. Not to mention queue sizes and network syncronization, and the always-looming threat of a buffer overflow.

9.4   The Network Client

This is the Meat and Potatoes of the Networking system. It is a static singleton class, and handles all incoming/outgoing messages, user command processing, creating/closing connections, and anything else related to networking.

There are many public utility functions, such as startGame, endTurn, and generateU-

niqueID. Most method calls, however, are to the send method, and its varients (send-ChatMessage, poke). This sends a message including a flag, tag, the username of that client, and any data passed to it through var-args. This allows for quite a bit of flexibility in calls, and leaves the processing of a message up to its arguments, not the method call itself.

Most of the difficulty in the implementation of the client was with command/message processing. The command thread processes any user input to the console box, such as network commands or chat messages, and sends the relevant messages. That was just a very hairy if-else mess, but not too terrible beyond constant maintence whenever a command changed.

The ClientRecieverThread, on the other hand, was troublesome. Its structure is almost an identical copy of its twin in ClientObject server-side. This meant a LOT of maintence whenever something changed, and lots of hairpulling resulted from the errors resulting from these subtle differences. This probably could've been done with a fancy design pattern during the design phase, but it was overlooked, and that led to a lot of extra time spent implementing.

9.5 The Network Server

This is actually simpler than the client side, even though it is utilizing more classes than the client side.

The Network Server itself is a static singleton class, with a constantly-running loop that listens for and recieves client connections. It manages the lobbies as a list of Lobby objects, and connected clients as a list of ClientObjects (each with their own pair of threads).

The primary functions of the server, beyond just passing along messages, is handling the "nitty-gritty" details of the network system. These include global broadcasts, connections/disconnects, and the mangement of lobbies. It does not care about the game state in any way.

That is the lobby's job. Each lobby object manages its own pool of clients, with a lobby having at least one client at any point in time. Lobby-wide broadcasts are handled here, as well as phase changes and game turns.

Each client connection that is recieved is assigned to a client object, each with its own name (players username) and unique ID. These objects handle all of the server's incoming and outgoing communication, and anything being sent out must go through a client object. Thusly, broadcasts involve looping through the client lists and calling the send function on each one. Additionally, they have the responsibility of processing ALL incoming messages, and calling the requisite methods in NetworkServer or one of the Lobby objects.

9.6 The Conductor

The Conductor is really just that: a "network train" conductor Anything game related that is the clients responsibility is passed to the conductor's processMessage method, which then invokes the requisite methods by using static instances of UnitPool, HUD-Controller, and others.

In its most basic form, its just a big switch statement that either directly calls the methods, or calls class utility methods that then directly call the methods on the instances.

The only major issue with it is the same as with the recieving threads: any command change requires mantence accross the entire network codebase. Not very modular, and very very brittle.

In spite of that, it does work, and serves its purpose in this instance.

10 The Map File Format

The MainMap data is stored in a large XML file

It has the following information including

- Hex ID
- Whether it's a capital, town, city, or castle hex
- Whether it's a town hex
- Neighboring Hex IDs
- Province Name
- Whether it's a vortex or portal
- What sorts of hex edges there are
- Whether the hex has a name

11 The Movement Calculator

The MovementCalculator class has 12 methods and two static fields. The static fields are for optimization of movement and retreating. The optimization of movement uses the allowanceCache HashMap to hold only the fast routes that the method getValidMoves(...)

finds. getValidMoves(...) is a public static method that recursively populates an ArrayList of MapHex objects based on constraints that apply to the unit that is moving. These constraints include terrain types, hex edges, enemy occupation and zone of control, and the movement allowance allotted to the unit per turn. The hashmap that optimizes the getValidMoves(...) method holds each valid MapHex that the unit can reach as a key and the unit's remaining movement allowance for moving into that hex. This allows a dynamic movement system, where a player may move a unit twice or more in a movement phase, while the unit still remains in the bound of its movement allowance. The other field in MovementCalculator, retreatAllowanceCache, serves a similar purpose as allowance cache, but is used by the retreat methods. Retreating is also done recursively, and all valid moves for a retreating unit are generated and added to the hashmap, retreatAllowanceCache. From there, the hash map is filtered according to the S&S rules for retreating from combat. There is a stated hierarchy of locations that a unit can retreat to, which the method filters for and follows, ultimately generating and returning a list of hexes that a unit can retreat to. If the size of the ArrayList returned is zero, or if the ArrayList is null, then the unit has no valid retreat locations, and should be destroyed. The remaining methods that are undescribed here are simply helper methods for the main methods, or wrappers, as in the case of wrapperMovement(...).

12 Gameplay Model Classes

12.1 Army Units

When implementation came around we discovered that having sub-classes for every different army unit sub-type was not practical. Instead a single army unit is with different flags to describe the unit is used. The main super class of movable units is still used with the sub-classes army unit, character, and monster but then stops there. Also a race and nation variable unit had to be added to the class in order for things like the movement calculator to work.

12.2 Characters

Characters have a series of variables needed for spell casting, leadership, and diplomacy. Something new is the idea of a spell book, a class that reads it's character's information to determine what spells can be cast. In order for this class to work properly the character class contains a spell book and the spell book contains its corresponding character. In order for the scenario reader to work properly a character creator was made in the character class that takes the name of a character, finds the matching data on the character, and creates a character with the data.

12.3 Spells

For spells to work properly they needed to exist separate from the main program. This was implemented by creating a separate GUI for spells. When a character "casts" a spell, the spell book class is called upon which created a GUI with all spell options that character has. Once a spell is chosen the user has the option of reading the details of the spell or casting the spell.

Once a spell is "cast", the specific spell class will be called and ask to select target(s) to cast. After a target is selected, limitations will be checked to see if succeed to cast the spell or not. If succeed, the effects of the spell are added to the games state. If not, there are two choise: "Recast" depending on different limitations or "OK" to terminate current spell implementation. Whatever the spell is cast or not, options that cast other spells or end spell phase are available.

12.4 Combat

12.4.1 LaunchCombat

The LaunchCombat functin first takes inputs Movable units, this connects with the HUD-Controller class: left mouse click to get selected stack, and right mouse click to get target stack. When a player in the combat phase, they should pick both units then press $'A'$ to launch the combat. When clicked A button, first it shows dialogs to ask the attacker player if they want to add friendly units which surround the defender units into the combat, and then it will pop out units detail include Attackers strength, Defenders strength, Defenders Terrain type, and Defenders strength after Terrain Type bonus with a confirmation dialog. The certain player should click yes button for showing combat result, or click no for doing nothing. If the player clicks yes button, the Combat result will shows as a notification on the right bottom conner, with this result both attacker player and defender player may meet three situations: Nothing Change, Retreat, Elimination. If the combat result returns 0, then the certain player does not need to do any reactions from this combat, if the result is a negative number, the certain player should eliminate numbers(since result is a integer number) of units from the units list; if the result is a positive number, then the player should decide to eliminate the unit OR retreat the unit.

12.4.2 Retreat

First the result came from the LaunchCombat, so the player can get its result value(negative, positive integer, or zero), if the result is a positive number then the player should decide to retreat the units or not, if not, then they should eliminate the unit, if yes, execute the Retreat function. The retreat function will send necessary input for the method getRetreatMove in the class Movement Calculator, so it can get which hexes that the certain

units can move to. And it will highlight those available hexes with red color, then the units will be randomly assigned to move to an available hex.

12.5   Unit Pool

Unitpool wasn't in the design.

It was added as a design modification during one of our sprints. The rules of the game required tracking of units, manipulation of persistent units, a way to create, and destroy those units. The class name is UnitPool.

12.6   HexStack

HexStack was originally going to be named Stack, and a component of a Hex.

In the implementation it exists to aggregate units in the UnitPool.

It's named HexStack instead of Stack to avoid a name conflict with the Java library.

Chapter 1

TEST PLAN

Version 1.1
May 9, 2014

FOREWORD

FOR

Swords and Sorcery

Prepared for:

Software Engineering: CS 383
Dr. Jeffery

Prepared by:

Keith Drew, Ian Westrope, Jonathan Flake, Colin Clifford, David Klingenberg, Sean
Shepherd, Chris, Gabe, John, Wayne, Tao Zhang, Cameron Simon, Matthrew Brown, Tyler
Jaszkowiak, Ian Westrope, ChiHsiang Wang

University of Idaho
Moscow, ID 83844-1010

RECORD OF CHANGES (Change History)

Change Number	Date completed	Location of change (e.g., page or figure #)	A M D	Brief description of change	Approved by (initials)	Date approved
1	April, 2014	Original HUD team document.	A	Original HUD team document.	HUD Team	April, 2014
2	May 5, 2014	Entire Document	M	Concatenated all team documents together.	DRK	May 5, 2014

A - ADDED M - MODIFIED D – DELETED

Swords and Sorcery Alpha 0.5

TABLE OF CONTENTS

1 TEST PLAN IDENTIFIER

TestPlan 1.5

2 REFERENCES

Use cases and UML diagrams are available on the class website, as well as game rules.
http://www2.cs.uidaho.edu/ jeffery/courses/383/
The code itself can be found at https://github.com/cjeffery/sworsorc/tree/master/src
The automated unit tests are located at https://github.com/cjeffery/sworsorc/tree/master/src/test

3 INTRODUCTION

3.1 HUD View

Our plan is to integrate the working hud/game code and test that it works in tangent, as opposed to only working in discrete locations of the project. We will be using junit tests to evaluate discrete methods and manual tests to test the GUI and game logic. To plan and document manually test for GUI components as they are developed. As new code is integrated the unit tests will be rerun to ensure integration did not break any component. Manual tests will be rerun on a case-by-case basis. All tests will be run the day before the end of a sprint and before deliverable presentation.

3.2 Rules View

The purpose of this test plan is to state the processes used by Game Rules and Play Team in testing the Spells and Characters for the Swords & Sorcery project.(Tao, Cameron)

This test plan covers the creation and implementation of the Moveableunit class and its sub class Armyunits. (Matt)

This test plan is also to provide as much coverage as possible to the methods and data of the Scenario class by verifying the results of execution against expectations through a series of automated unit tests and a manual GUI test. (Tyler)

3.3 Network View

This test plan for the networking portion of the project. It will outline how we test networking, and how we'll test integration of networking into other code.

4 TEST ITEMS

The army units will be tested for the proper member variable values as well as proper results from the member function of the Moveableunit and Armyunit classes. Variables will tested at creation and members variable will be tested for proper results when applicable. Some example tests will be that the location member variable is properly set during movement. Also another type of test that will be preformed is that conjured units are properly created and and placed in the proper place on the game board.

4.1 Character

- Class Interfaces

- Class Interactions

- Spells Implementation

- Character

4.2 Scenario

The data read by the Scenario test will be read from one of the simple scenario configuration files and then verified against expectations through the class's accessor methods. These data items include

- The scenario's name

- Number of players

- Game length

- The blue sun's initial position

- The names of armies in the scenario

- The controlling players of these armies

- The setup order of these armies

- The movement order of these armies

- Names of nations within these armies

- Names of the neutrals in the scenario

- The provinces controlled by a nation

- The characters within a nation

- The units within a player nation

- The units within a neutral nation

- The races of both neutrals and player nations

- The reinforcement and replacement description strings

- Data related to where a neutral is leaning toward

- Whether or not a neutral accepts human sacrifice

Unfortunately, the most complex functionality of the Scenario class cannot be tested by automated unit tests. The unit pool populator requires a manual check.

4.3 Network

- Client connects and disconnects from the server.
- Server detects new client connections.
- Client and server can exchange messages.
- Client and server are able to identify message type and content.
- Client messages are received by related clients.
- Network events trigger GUI events properly.

4.4 GUI

- Hexes/Units tile ok, look ok
- Mouse clicks work
- Menu navigation
- Game starting (networked, scenario)
- General gameplay (to the extent it's implemented)
- Unit movement
- Diplomacy map displays thru menu bar selection
- MouseEvents on main map correspond to correct hexID
- Selecting unit/stack works with left mouse click
- Selecting a unit/stack displays unit information in correct pane
- Deselecting unit/stack works with left mouse click after unit/stack has been selected.
- Targeting unit/stack works with right mouse click
- Targeting unit/stack displays unit information in correct pane
- Detargeting unit/stack works with right mouse click after unit/stack has been selected.
- Selecting a target is not possible during movement phase
- Move unit works with right click once a unit has been selected
- Movement calculator correctly displays the available locations to move to
- If friendly and target units are selected during combat phase, 'a' will start combat
- If friendly and target units are selected during the spell phase, 's' will cast spell
- Keyboard Events don't affect the message box when message box is not selected
- Message box clears itself after every message is entered
- Chat box displays messages from others on the server
- Sun positions and display only update after every game turn
- Initial sun position corresponds to that in the scenario

4.5   Additional items to be tested.

- Other manual tests (game compiles and runs on different platforms)

- Auto tests - Junit

- Ensure all files under resources can be loaded (if feasible)

5   SOFTWARE RISK ISSUES

Software to be tested includes the following:

1. User's network configuration prevents networking.

2. GUI - Doesn't load working map, can't support unit movement

3. JSON Library - Loads scenario incorrectly, if not at all

4. Incorporating Networking with GUI and game logic may be difficult

5. Undetected Logic Errors (ULEs)

6. Misinterpretation of Rules

7. Connection Failures: For whatever reason, we have no access to a server, or can't connect over user's network

8. Networking code is unable to connect with other code elements.

9. Networking code can be integrated, but is too complicated for anyone to work with.

10. Improper casting of units

11. Unit ID's interpreted incorrectly

12. Depends on Java's JSON reader and the programmer's understanding of it

13. Complex data structures such as a map of maps

14. Complex loops to iterate over data structures

15. Poor documentation surrounding some of these iterators

16. Poor documentation in general

17. 3rd party library's. Including json-simple-1.1.1 and controlsfx-8.0.5

6 FEATURES TO BE TESTED

6.1 Scenario

From the user's perspective, much of this data reading occurs under the hood. In all cases except for populating the unit pool, the Scenario class does not manipulate any pieces of the rest of the project. Other components read the data from the Scenario class. Therefore, while the solar configuration relies on a properly-working Scenario class, this is not apparent to the user because solar configuration is also handled by SolarConfig and HUDInitializer classes.

Therefore, the only visible component being tested by this plan is the placement of units and characters into the correct provinces of the map.

This is a listing of what is NOT to be tested from both the Users viewpoint of what the system does and a configuration management/version control view. This is not a technical description of the software, but a USERS view of the functions.

All data is verified, but testing the Scenario class alone cannot ensure that it reaches the HUD successfully. Integration tests are required outside of this plan for information such as solar configuration, move order, setup order, game length, and diplomacy.

What is not tested is the count of the units on the map that correspond to the scenario loaded. Also the type of unit is not test, this is assumed correct.

6.2 Movement

- conjured unit appear when casted
- units are properly represented on HUD
- movement location is correct on map
- moral status is properly updated after combat

6.3 Character, Spells

- Select character on GUI
- Select Spell
- Effects of Partial Spells
- Manna Costing

6.3.1 Character

Rule Description	Test Description	Expected Result
Create new character object with its information.	createCharacter function is called with specific character name in CharacterMaker.java.	A new character object is created and returned that contains all of the specified characters information.

Potential spells for selected character displayed.	Select character in GUI, then click cast spell button on sidebar panel.	List of spells that can be cast by selected character are displayed.

6.3.2 Spells

Rule Description	Test Description	Expected Result
Character with Power Level should have a spell book.	Generate a spell book for the character.	A frame with a list of spells should be shown on the screen.
Show spell description.	Click on the Spell button.	A frame will be displayed with all information about the spell.
A target need to be selected for most of the spells.	Click on cast button on the frame of displaying information about the spell. Then select a target by right click the target on the map.	A target is selected to cast the spell.

6.4 Additional Features To Be Test

- 1 Complete Turn

- Movement

- Teleporting

- Network

- Loading

- Solar Display

- Diplomacy Display

7 FEATURES NOT TO BE TESTED

All data is verified, but testing the Scenario class alone cannot ensure that it reaches the HUD successfully. Integration tests are required outside of this plan for information such as solar configuration, move order, setup order, game length, and diplomacy.

What is not tested is the count of the units on the map that correspond to the scenario loaded. Also the type of unit is not test, this is assumed correct.

- Full game

- Everything in backlog

7.1 Network

JUnit tests will be used to test both sides of the network (that is, the client and the server networking code), in isolation.

Manual testing will be needed to ensure integration of the networking code with elements of the graphical user interface, as well as testing that the client and server work over physically distinct machines. Because of the limited resources for manual testing, manual testing will test a subset of possible interactions, with the assumption that core network obstacles will block all messages (i.e. no connection can be made over the network), or no messages. The graphic effects of network events will also be tested manually.

8 APPROACH

The plan is to use Junit tests and manual tests to ensure that our code follows the rules of the game and works itself. Junit tests are done according to the individuals who have developed the methods being tested, and those are not listed here. However, they should be able to be run together and work. The manual tests will work as though a mock player (tester) is running the game. They should be able to select a unit, move a unit, teleport a unit, advance and view the solar display, send chat messages, and view the diplomacy display. Functionalities of each display should also be tested. For example, a unit with a move allowance should only be viewed moving at or under their limit and a chat message should be sent from one user and be seen by all users.

8.1 Network

JUnit tests will be used to test both sides of the network (that is, the client and the server networking code), in isolation.

Manual testing will be needed to ensure integration of the networking code with elements of the graphical user interface, as well as testing that the client and server work over physically distinct machines. Because of the limited resources for manual testing, manual testing will test a subset of possible interactions, with the assumption that core network obstacles will block all messages (i.e. no connection can be made over the network), or no messages. The graphic effects of network events will also be tested manually.

9 ITEM PASS/FAIL CRITERIA

All tests should meet the specifications of the Swords and Sorcery board game rules manual, as well as follow logical design patterns and language rules/conventions. The manual tests of movement should fall within 20% of total movement possibilities the movement rules indicate.

10 SUSPENSION CRITERIA AND RESUMPTION REQUIRE- MENTS

- If the github project is broken or otherwise compromised, productive development and testing would be suspended for a time. To resume, fix whatever has been broken on github.

- If any component test fails, the larger dependent pieces are unable to be tested until the component is fixed. In such a case, fix the component and continue testing.

11 TEST DELIVERABLES

- Test plan document

- Test cases

- Relevant error logs or problem reports

- Possible solutions

12 REMAINING TEST TASKS

There are many remaining test tasks, as we have not achieved a working game yet. Remaining tests include full combat, spell casting, scenario loading, full gameplay, etc. We also anticipate unexpected integration requirements to be added to the test plan.

13 ENVIRONMENTAL NEEDS

In the final test, we'll need a remote machine to host the server code. Along with two computers connected over the internet too the server. Otherwise, we cannot test full networking capabilities and full gameplay as intended.

14 RESPONSIBILITIES

The hierarchy of "inchargeness" is as follows:

1. Dr. J

2. John Goettsche

3. Everyone else

15 SCHEDULE

Junit tests should be implemented and run as functionality is added to classes and packages. Manual testing will be done as the GUI is developed and functionality added, as well as when more aspects of other group's code are added. Test to be rerun before demonstration day.

16 PLANNING RISKS AND CONTINGENCIES

As the end of the semester is a fixed date, there are no contingencies for failure. Public beatings will be carried out as needed.

17 APPROVALS

The only person capable of fully approving any fixes/modifications is Dr. J.

18 GLOSSARY

Junit test is a discrete code test designed to be ran as part of an automated test sequence that tests all Junit tests.

19 APPENDIX A. Example JUnit Test

```java
@Test
public void testStackWarning() {

        boolean test;
        pool.addUnit(1, new Bow(), "0101");
        pool.addUnit(1, new Bow(), "0101");

        test = HexStack.overStackWaring(pool.getUnitsInHex("0101"),false);

        assertFalse(test);

        pool.addUnit(1, new Bow(), "0101");

        test = HexStack.overStackWaring(pool.getUnitsInHex("0101"),false);
        assertTrue(test);
}
```

20   APPENDIX B. Example Manual Test

This test will ensure that the graphical components of the over stack warning in the stack class is behaving as designed.

1. Start with any hex that has one unit in it.

2. Add one unit to the stack. No warning should be triggered.

3. Add one unit to the stack a warning should be triggered.

4. Close the over stack warning box.

5. Remove one unit from the stack. No warning should be triggered.

6. Remove one unit from the stack. No warning should be triggered.

7. Add one unit to the stack. No warning should be triggered.

8. Add one unit to the stack a warning should be triggered.

9. add one unit to the staff a warning should be triggered indicating there are 2 too many units in the stack.

The test pass if all steps are successfully completed. Otherwise, test fails.

21 APPENDIX C. Individual Team Test Plans

All individually developed tests to be found at the following link:
    https://github.com/cjeffery/sworsorc/tree/master/doc/TestPlans

Metrics Report
Including Analysis of Coverage, Complexity, Cohesion, and Coupling
For
Swords and Sorcery(S&S)

Version 1.0

Prepared by:
University of Idaho Computer Science 383 Class, Spring 2014

Prepared for:
Dr. Clinton Jeffery

May 9, 2014

Swords and Sorcery Metrics
Table of Contents

1   Introduction

This document includes reports and analysis of several software metrics of our computer adaptation of the Swords & Sorcery board game. The project and its documentation are property of the Spring 2014 Software Engineering class at the University of Idaho and its constituents.

Through this document, the development team aims to describe the software using several measurements such as Test Coverage, Complexity, Coupling, and Cohesion. Analysis of these metrics and what they mean for the software follow.

2   Test Coverage

The project's 79 automated test cases resulted in 9% instruction coverage and 5% branch coverage of the entire project. While the average is low for the whole project, some elements exhibit higher levels of coverage.

The leader in instruction coverage was the MoveCalculator package, which experienced 53% instruction coverage (and 44% branch coverage). Meanwhile, the most branch coverage was attained by the ssterrain package, at 53% (and 48% of instructions).

The top level coverage data is summarized by the table below.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
MoveCalculator		53%		44%
ssterrain		48%		53%
sshexmap		33%		29%
Units		25%		12%
systemServer		8%		3%
sscharts		4%		1%
mainswordsorcery		0%		0%
CombatSimulator		0%		0%
Spells		0%		0%
Character		0%		0%
buildmapfilel		0%		0%
Spells.PL_6		0%		0%
Spells.PL_3		0%		0%
Spells.PL_2		0%		0%
Spells.PL_4		0%		0%
Spells.PL_1		0%		0%
default		0%		0%
Spells.PL_5		0%		0%
unitdetailsdisplay		0%		0%
Spells.PL_7		0%		0%
ObjectCreator		0%		n/a
phase		0%		n/a
Total	50,483 of 55,448	9%	6,570 of 6,929	5%

Ignoring percentages, the package with the most covered instructions was ssterrain with 1,374 covered instructions. Of its 39 classes, only 10 were missed by the automated unit tests. All hex edge types had 100% coverage except for HEFord. While the various terrain types were not covered well, TerrainType itself had 94% instruction coverage.

You'll notice from the table that the sscharts package sticks out sorely as far as coverage goes. This is due largely to the very large and completely untested class CreatUnits, which has 14,363 instructions. This is demonstrated below.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
CreatUnits		0%		0%
ArmyCombatResultsTable		0%		0%
RandomEventTable		0%		0%
Scenario		69%		55%
Total	16,158 of 16,771	4%	3,298 of 3,320	1%

Whilte sscharts gets a bad report due to one untested monster, the systemServer package warrants its bad marks by leaving several significant classes untested, as shown by the following.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
ClientObject.ServerReceivingThread	▰▰▰▰▰▰	0%	▰▰▰▰▰▰	0%
ClientObject.ServerListenerThread	▰▰▰▰▰▰	0%	▰▰▰▰▰▰	0%
MessageUtils	▰▰▰▰▰	0%	▰▰	0%
NetworkClient.ListenerThread	▰▰▰▰	0%	▰▰▰▰	0%
NetworkClient.ClientReceivingThread	▰▰▰▰	0%	▰▰▰▰▰	0%
NetworkClient.ClientListenerThread	▰▰▰▰	0%	▰▰▰▰▰	0%
ClientObject.ListenerThread	▰▰▰▰	0%	▰▰▰▰	0%
NetworkClient	▰▰▰▰	25%	▰▰▰▰	12%
NetworkClient.ClientCommandThread	▰▰▰	0%	▰▰▰▰▰	0%
NetworkServer	▰▰▰	39%	▰▰▰▰	27%
ClientDataForm	▰▰	0%	▮	0%
ClientObject.new Object() {...}	▰▰	0%		n/a
NetworkClient.new Object() {...}	▰▰	0%		n/a
ClientObject	▰	0%	▰▰	0%
Lobby	▰	0%	▰	0%

The full coverage report can be accessed in the sworsorc repository as sworsorc/.jacocoverage/report.htm for full details.

sshexmap has 16 classes and 9 were missed. The best coverage was with the Hex class with 100% coverage, followed by HexMap with 78% and MapHex with 69%. Of the 140 methods in all the sshexmap classes 97 were missed and the 801 lines, 550 were missed.

Units has 291 classes and 234 were missed. The classes that had 100% coverage were Zeppelin, Bow, RangedLandUnit, LandUnit and FlyingUnit. Of the 156 methods in all the classes of Units 110 were missed and of the 922 lines 716 were missed. The UnitPool had 47% coverage.

Because test coverage in most cases was extremely poor for this project, we have done very little to assure that the componenets of the project perform as they are expected to. This means that very little can be said about the quality of the product. Our tests simply cannot conclude that it performs well. When we analyze the complexity of various components later, we will see that some of our worst coverage overlaps with our most complex methods. It becomes impossible to assure product quality when the components that are most difficult to understand are also unchecked.

3   Complexity

When setting out to to develop a semester project for this course, we knew that it would have to be complex enough to take a multi-person team of developers a semester to create. These complexity metrics serve the dual purpose of assuring the reader that this has occurred while also pointing out exactly how frightening it is that test coverage is so low.

3.1 Lines of Code (LOC)

The project totals 49,862 lines of code. These are broken up between several packages, with the following five having the highest LOC count:

- src: 40903

- model: 10485

- src: 10218

- CombatSimulater: 10218

- utilities: 9777

The following five classes are the longest as well:

- Main: 6597

- CreatUnits: 6575

- NetworkClient: 1281

- CharacterJ: 1222

- CharacterJ: 1222

You'll observe that despite breaking the project up into several modules, there are still some files that total over six thousand lines of code. This could degrade both the readability and maintainability of the code base, as it requires a great deal of effort just to understand the code of one class.

3.2 Class Count

Even with some classes being so voluminous, we still managed to acquire 303 total classes throughout the course of development. Once again, this can have a poor effect on the understandability of the project. The packages with the most classes are listed.

- src: 244

- model: 119

- src: 70

- CombatSimulater: 70

- doc: 59

3.3 Method Count

There are a total of 1614 methods. These are more evenly distributed through classes, as shown by the top five classes by method count. While this is still a lot of methods for a single class to contain, at least there are no outlying monsters in this respect.

- NetworkClient: 42

- CharacterJ: 42

- CharacterJ: 42

- MainMap: 35

- MapHex: 30

3.4 McGabe's Cyclomatic Complexity

While the average cyclomatic complexity of the project's methods was a comfortable 3.33, outliers emerge that are purely terrifying. The unchecked CreatUnits class provides one of these, while the others are equally unchecked methods within Main.

- Main::Create_unit1: 252

- Main::Create_unit2: 252

- Main::Create_unit4: 252

- Main::Create_unit3: 252

- CreatUnits::Create_unit2: 202

These monstrously-complex methods include nested switch statements (thus the high volume of paths possible), and it is unclear what they are meant to do. This has a disastrous effect on readability, comprehension, and maintainability. Furthermore, something this complex and untested is likely to be broken, meaning that maintainability and quality are also affected in that sense.

4 Coupling

Our metrics tools gave the following measurements of coupling.

Loose Class Coupling (LCC) averaged 0.151 with a maximum of 1 and a minimum of 0 for the project. High Class Coupling (HCC) on the other hand also ranged from 0 to 1 and averaged 0.111.

From this data, we can conclude that about 15% of our classes are loosely coupled, meaning they require little knowledge of the definitions provided in other classes. This means that these classes may experience an increase in readability, maintainability, and testability because they can stand alone.

On the other hand, about 11% of our classes were tightly coupled, which means understanding, testing, and executing them requires knowledge of the definitions in other classes. This can be detrimental to maintainability, testability, and readability.

5 Cohesion

The five measure sof Lack of Cohesion had the following minimums, maximums, and averages for the project.

1. 32, 299, 43
2. 28, 273, 39
3. 2, 6, 5
4. 2, 6, 4
5. 0.925, 0, 0.553

From these metrics, we can make the following conclusions.

1. There is a minimum of 32 and at most 299 pairs of methods in a single class that do not share attributes. There is an average of 43 methods per class with low-cohesion.

2. There is a minimum of 28 and at most 273 methods after subtracting the number of pairs of methods that share attributes from the number of pairs of methods that do not share attributes.

3. There is a minimum of 2 and maximum of 6 disjoint components per class with an average of 5 disjoint components per class.

4. Similar to Method3 but measuring method invocations instead of node edges.

5. LCOM5 $\bar{}$ (a – kl) (l – kl), where l is the number of attributes, k is the number of methods, and a is the summation of the number of distinct attributes accessed by each method in a class. The average lack of cohesion across all our classes using LCOM5 is 55.3%

Unfortunately, some of these measures of Lack of Cohesion are fairly high, meaning we have fairly poor cohesion between our classes. Unfortunately, a lack of cohesion can mean that the methods and data in classes are unrelated. This has the effect of reducing readability and maintainability of methods. Additionally, since classes with unrelated elements are likely to be constructed in a design-specific manner that will limit their reusability.

6   Appendix

A report of several more metrics can be found as generated by the NetBeans Source Code Metrics plugin at sworsorc/doc/EndingDocumentation/Metrics/metrics.xlsx.