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 8, 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: Ä collection of products to document an architecture.¨ISO/IEC 42010:2007
Alpha Test	Limited release(s) to selected, outside testers.
Architectural View	Ä representation of a whole system from the perspective of a related set of concerns.¨ISO/IEC 42010:2007
Architecture	Ïhe 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	Än 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	Ä 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
GUI	Graphical User Interface - What the user sees and interacts with - also called the HUD.
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.
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.
PlayerID	An integer representing one of the factions described in this scenario.
S&S	Swords and Sorcery
Server	The (single) process that a client connects and sends messages to.
SSDD	System and Software Design Document
SSRS	System and Software Requirements Specification
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, textitSystem 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. 3) 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.

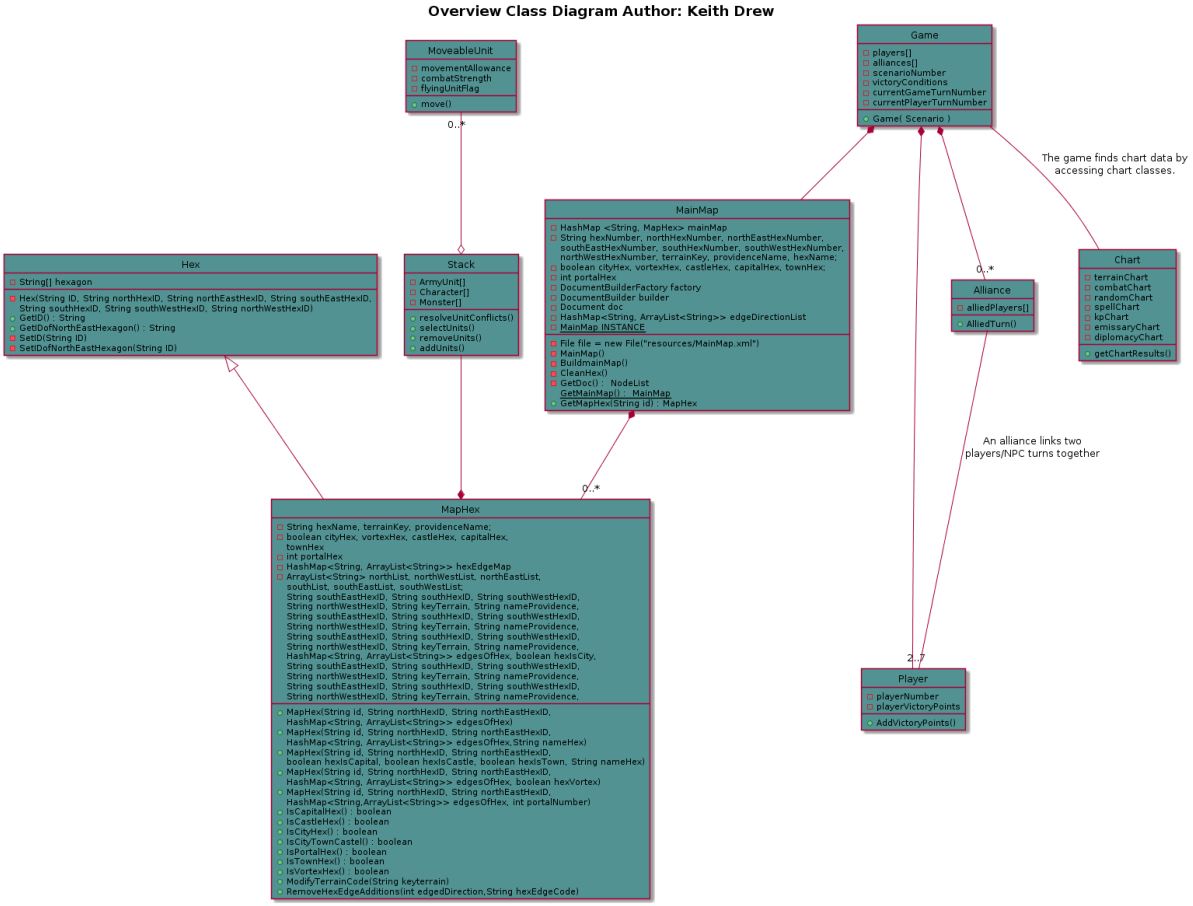
HUD Team View The HUD Team view includes all software design related to the HUD/GUI and handles how the user(s) interact with the S&S game rules and networking.

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.

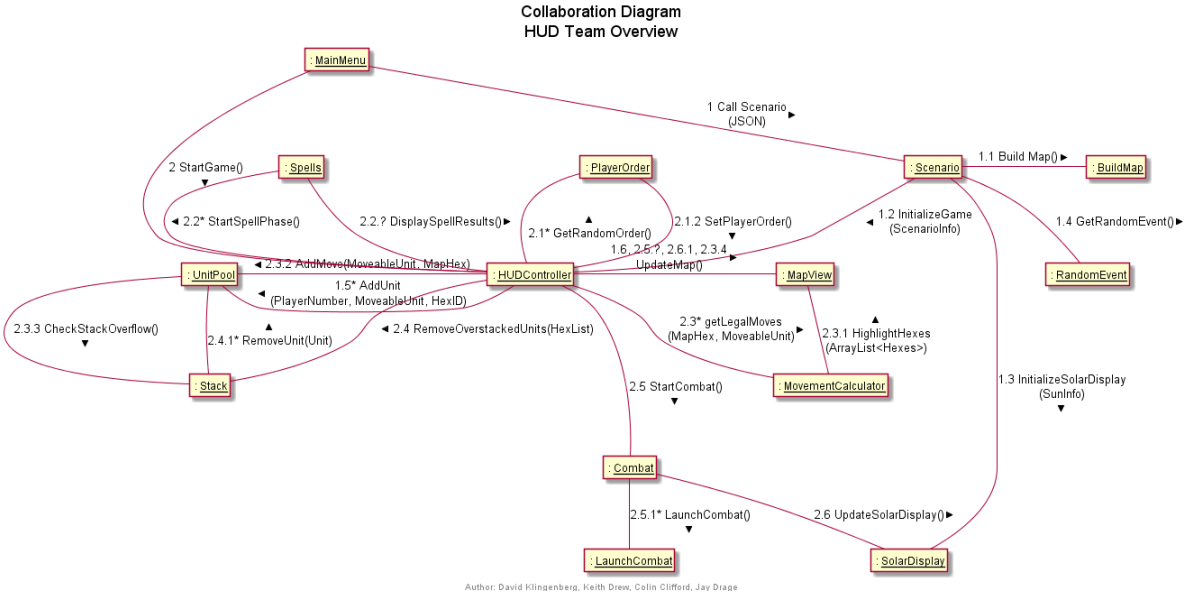
Networking Team View The Networking Team view includes all network communication related to the S&S game. This includes the client/server model, the communication protocols, the server setup and more.

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.



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.



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).

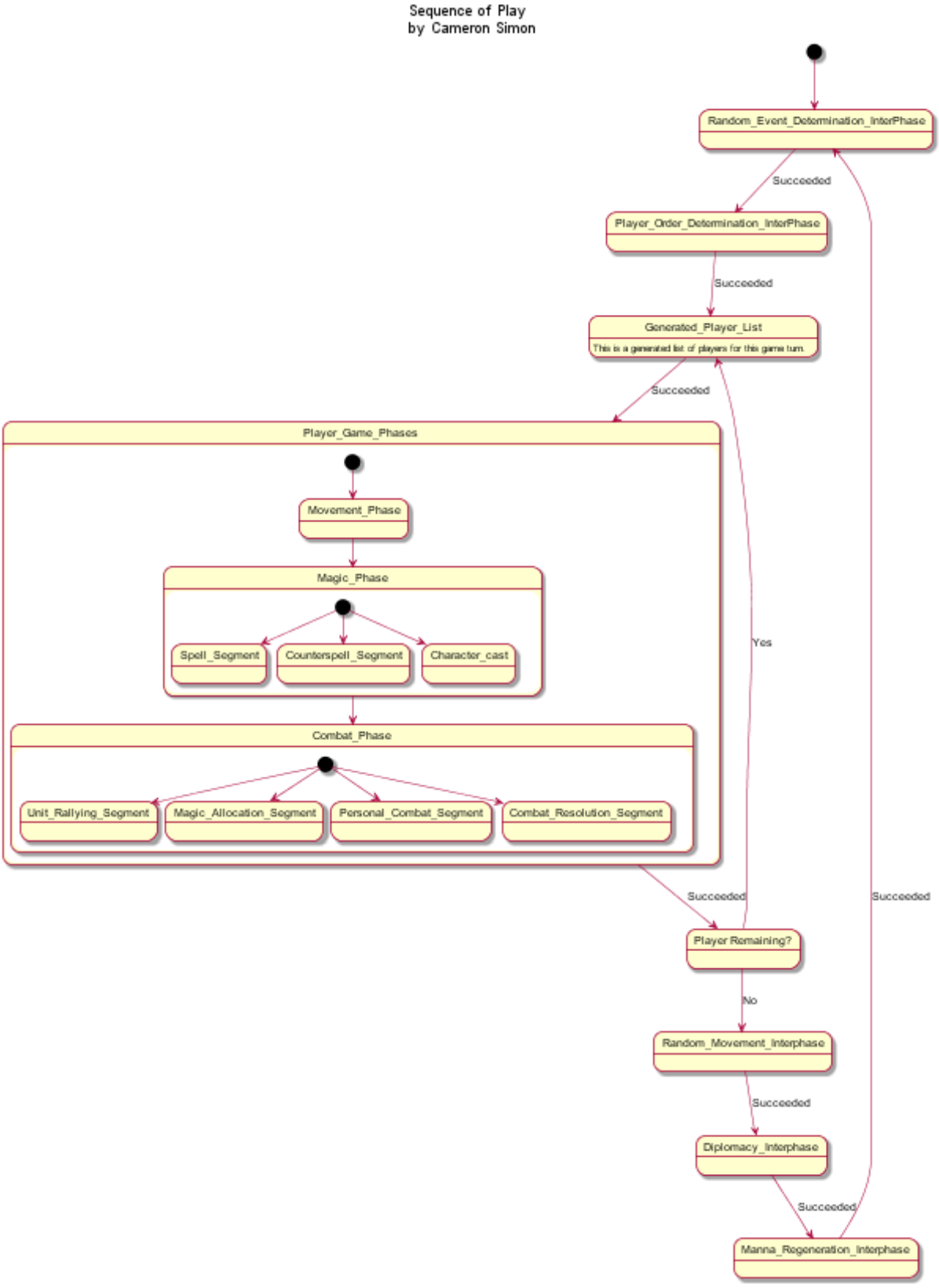
Networking Team View Representation and Description The networking team view covers the client/server implementation and design for S&S. Their overall view of design was a client/server model which hosts communication over the network and message passign for updating the state of a game.

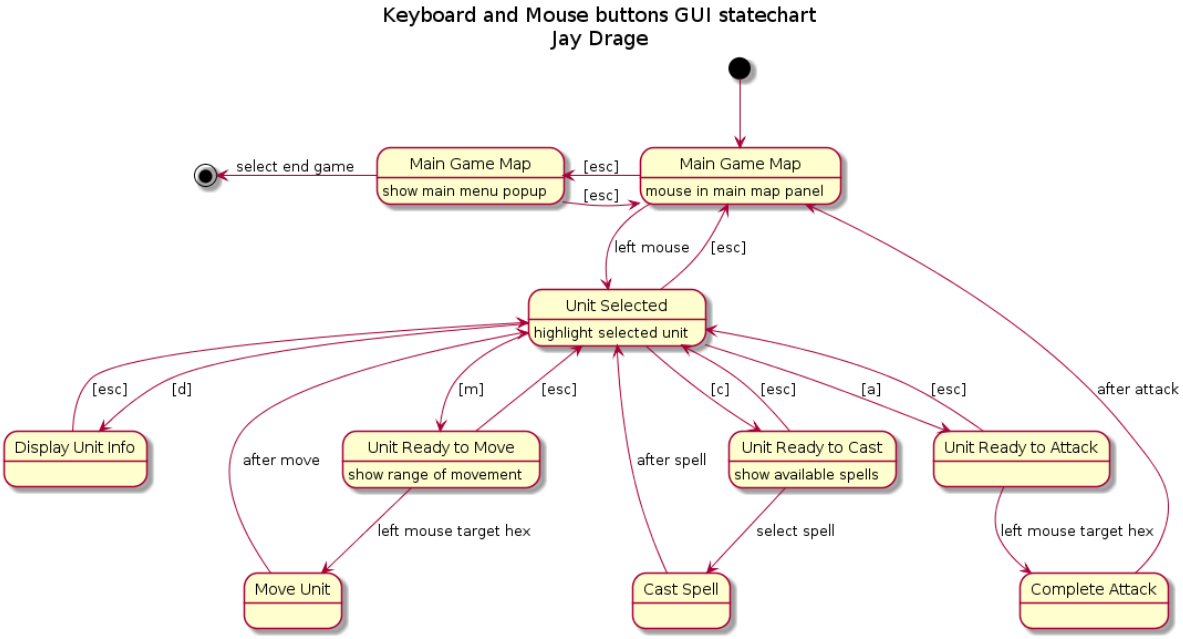
3.2 User's Architectural View

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



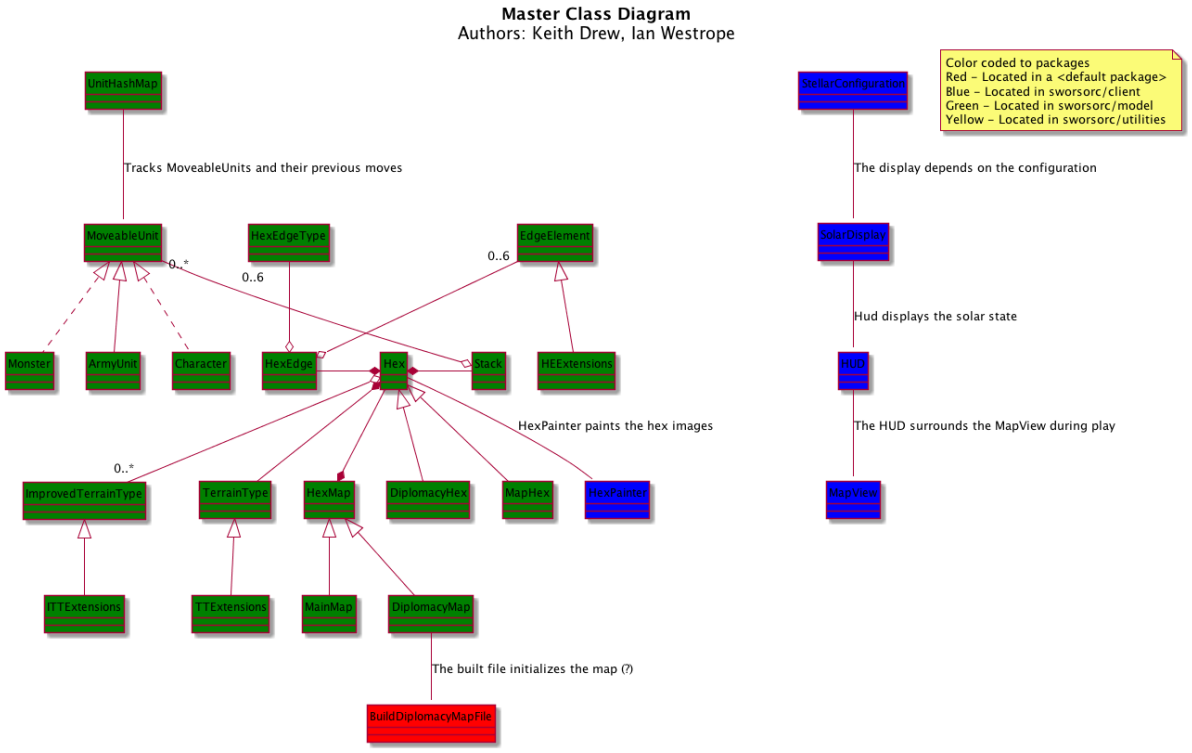

Keyboard and Mouse buttons GUI statechart
Jay Drage
select end game
Main Game Map
show main menu popup
[esc]
[esc]
Main Game Map
mouse in main map panel
left mouse
[esc]
Unit Selected
highlight selected unit
[esc]
[d]
[m]
[esc]
[c]
[esc]
[a]
[esc]
after attack
Display Unit Info
after move
Unit Ready to Move
show range of movement
after spell
Unit Ready to Cast
show available spells
Unit Ready to Attack
left mouse target hex
select spell
left mouse target hex
Move Unit
Cast Spell
Complete Attack

4 Software Detailed Design

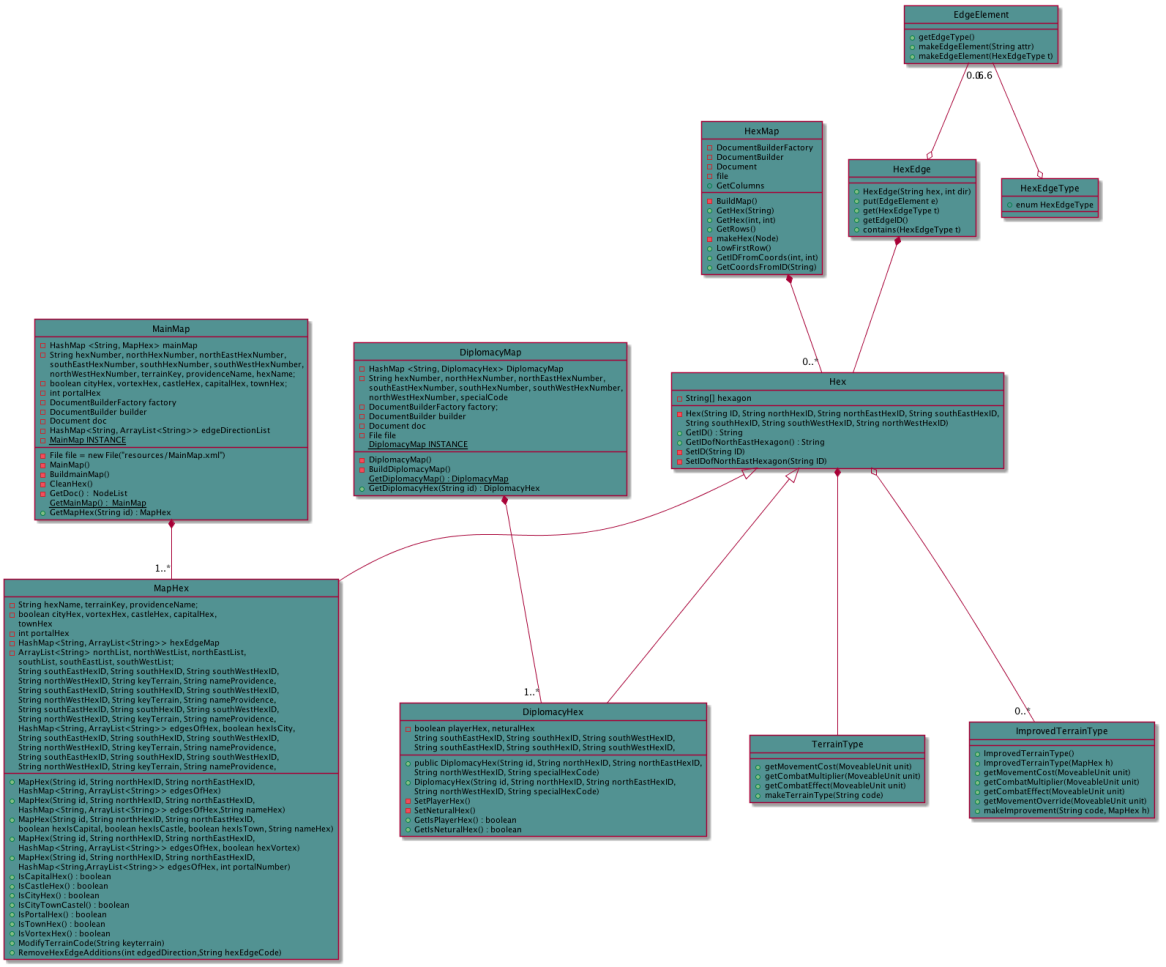
4.1 Developer's Viewpoint Detailed Software Design

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.

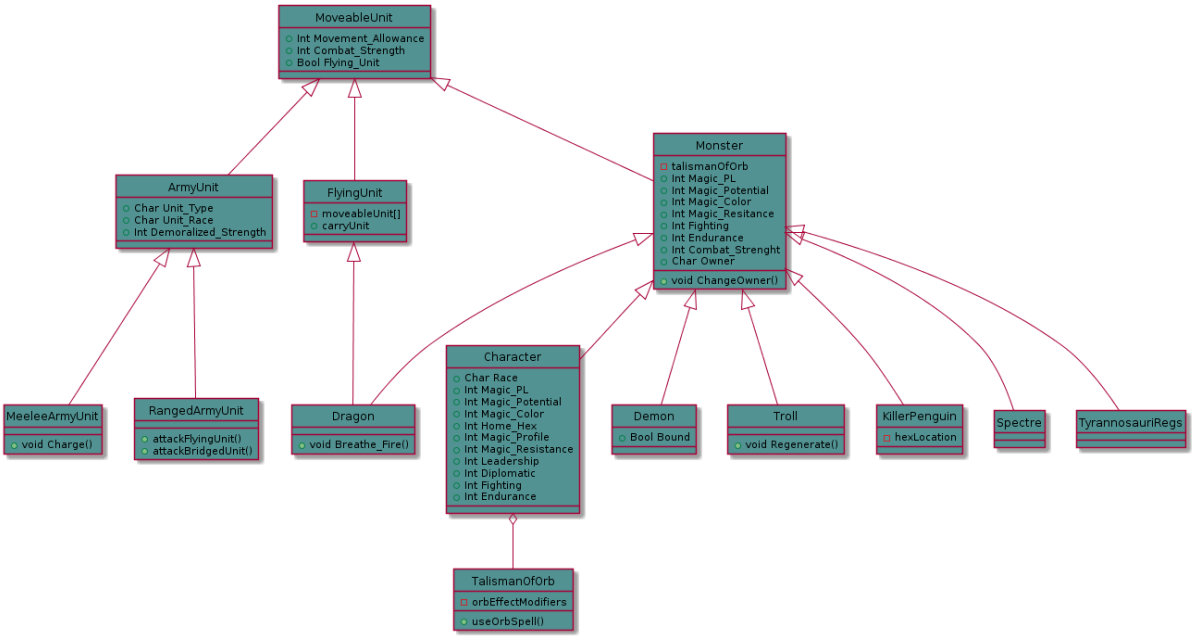
4.1.1 UML Class Diagrams

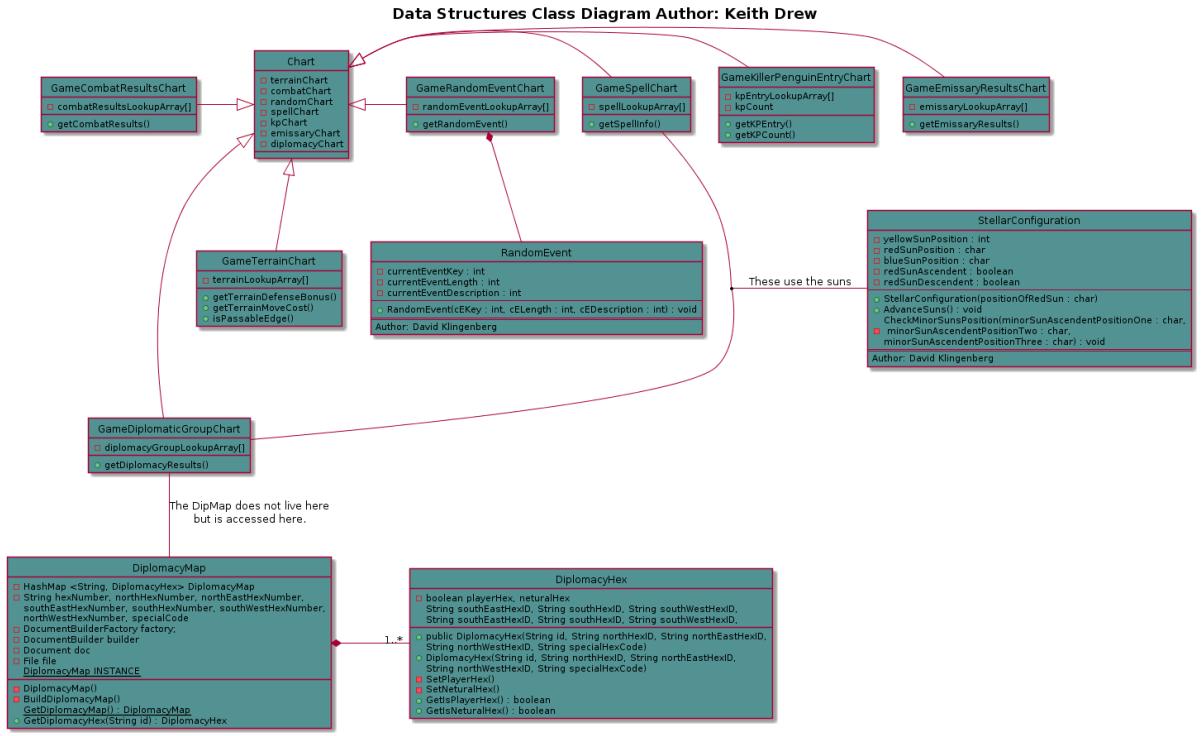


Hex Class Diagram Author: Ian Westrope, Keith Drew

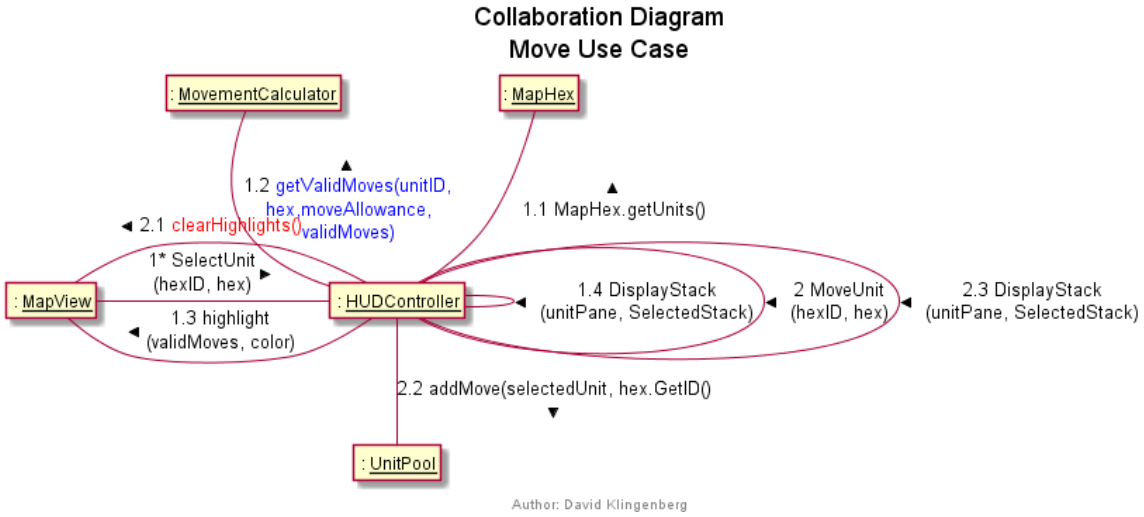


Unit Class Diagram Author: Ian Westrope

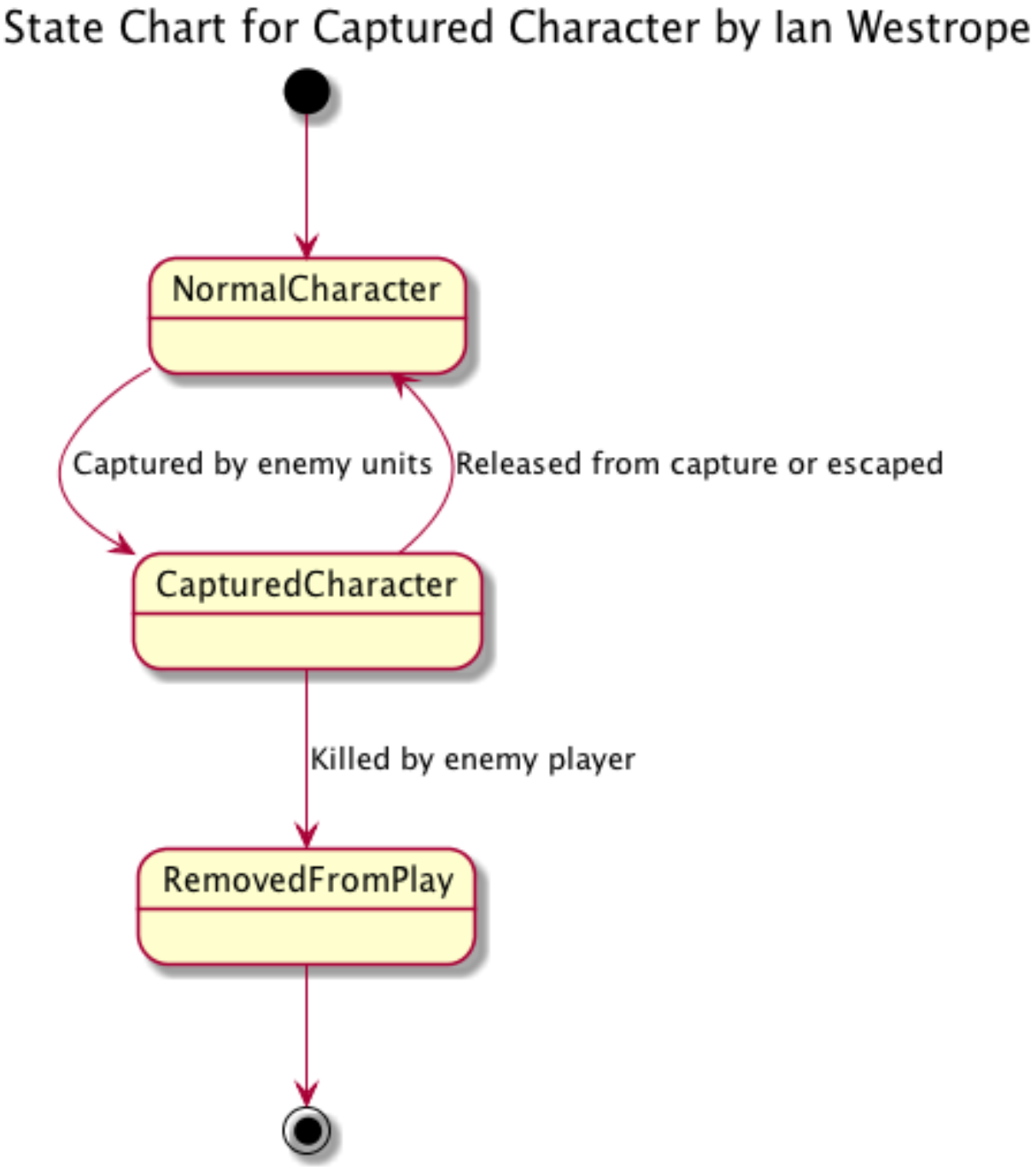


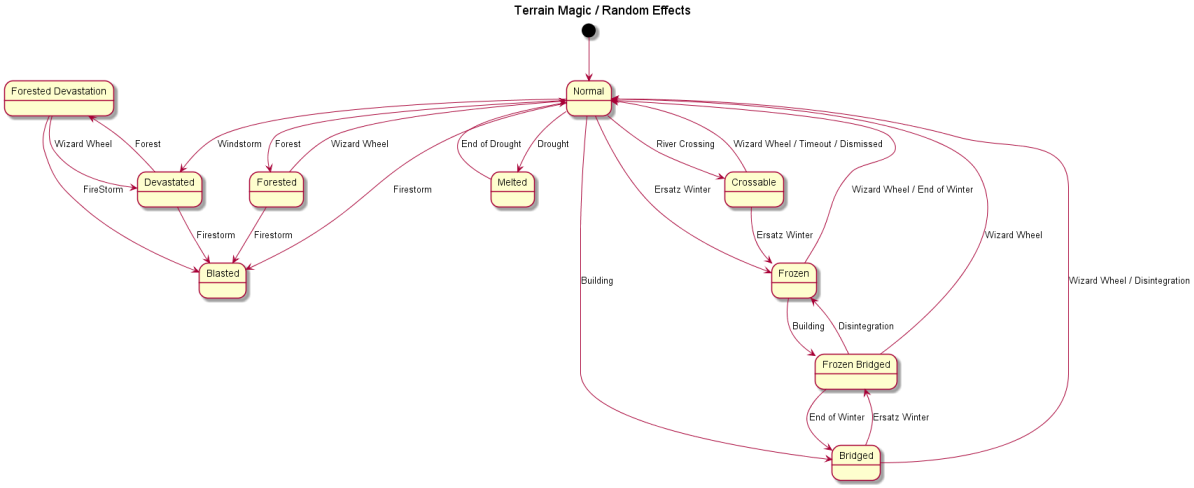

Data Structures Class Diagram Author: Keith Drew
Chart
terrainChart
combatChart
randomChart
spellChart
kpChart
emissaryChart
diplomacyChart
GameCombatResultsChart
combatResultsLookupArray[]
getCombatResults()
GameRandomEventChart
randomEventLookupArray[]
getRandomEvent()
GameSpellChart
spellLookupArray[]
getSpellInfo()
GameKillerPenguinEntryChart
kpEntryLookupArray[]
kpCount
getKPEntry()
getKPCount()
GameEmissaryResultsChart
emissaryLookupArray[]
getEmissaryResults()
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
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
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

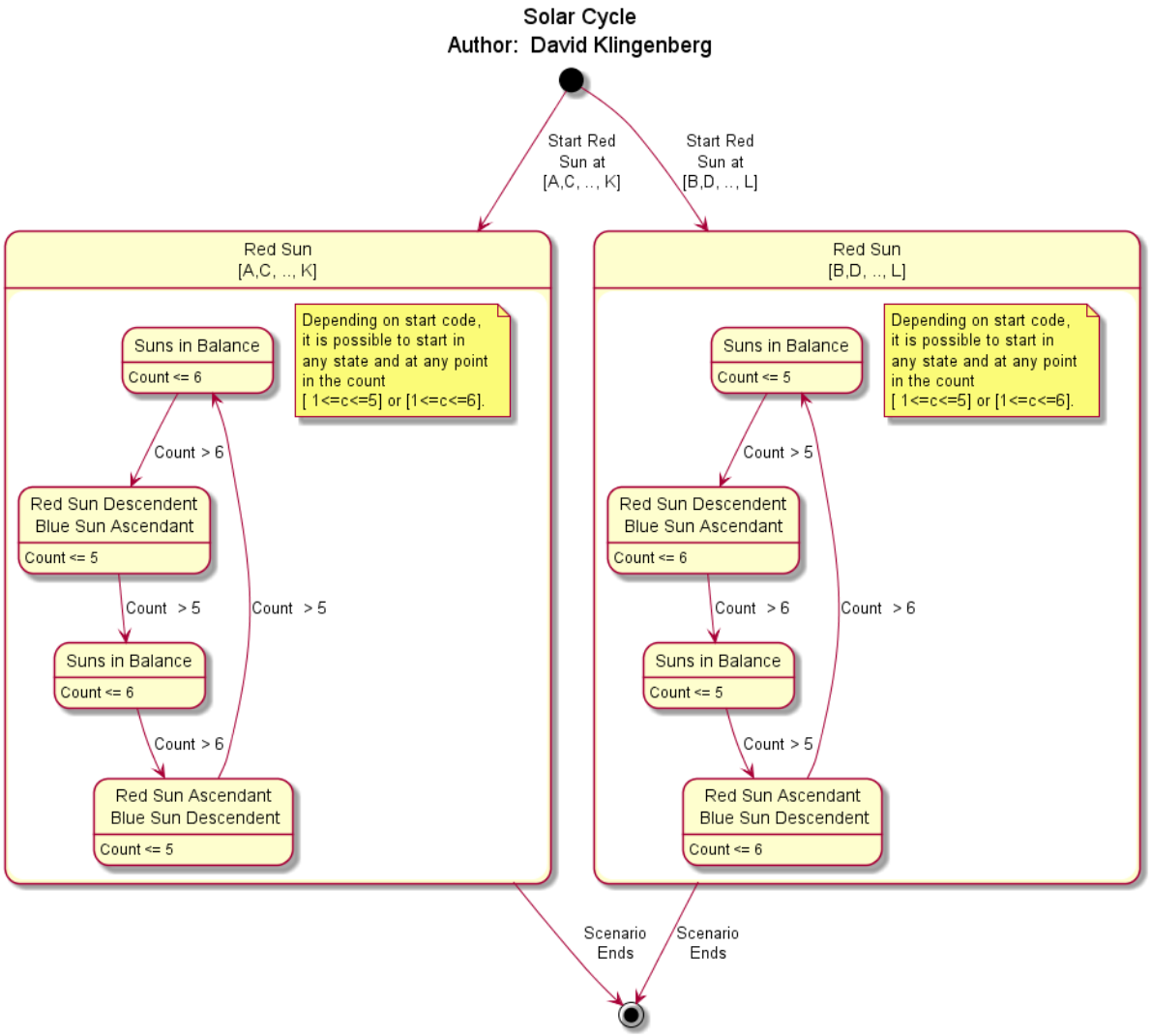
4.1.2 UML Collaboration Diagrams



4.1.3 UML State Charts




Terrain Magic / Random Effects
Normal
Forested Devastation
Devastated
Forested
Melted
Crossable
Frozen
Frozen Bridged
Bridged
Blasted
Wizard Wheel
Forest
Windstorm
Forest
Wizard Wheel
End of Drought
Drought
River Crossing
Wizard Wheel / Timeout / Dismissed
FireStorm
Firestorm
Ersatz Winter
Wizard Wheel / End of Winter
Firestorm
Firestorm
Ersatz Winter
Wizard Wheel
Building
Wizard Wheel / Disintegration
Building
Disintegration
End of Winter
Ersatz Winter


Solar Cycle
Author: David Klingenberg
Start Red Sun at [A,C, .., K]
Start Red Sun at [B,D, .., L]
Red Sun [A,C, .., K]
Red Sun [B,D, .., L]
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
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 overstacked array.	Sorted map: overStackMap	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	
EndMovementPhase	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.		
GetAllPlayerUnits	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: overStackMap	None
GetPlayerSpecificUnits	UnitPool Method	Get all units by type owned by a player.	Sort map: pool	None
GetSafeTeleport	UnitPool Method	Determine if a unit can safely teleport.	Array list: safeTeleport	None
GetTeleportDestination	UnitPool Method	Get a unit destination portal.	Sorted map: portalNum	None
GetUnit	UnitPool Method	Retrieves a units.	Sorted map: pool	None
GetUnitHexMove	UnitPool Method	Retrieves all hexes a unit has moved through.	Array list: unitMove	None
GetUnitInHex	UnitPool Method	Retrieves all units in a hex.	Array list: hexList	None
Hexagon Classes	Model	Represents a hexagon	Hex Edge Classes, Terrain Classes, UnitPool	
Hex Edge Classes	Model	Classes to collect and represent elements on hexagon edges	Hex Classes	
HexStack	Class	Ensure compliance with the games stacked rules.	UnitPool	None
HexStacked	Class	Implements rules for stacks.	UnitPool	None
Hexagon Map Classes	Model	Classes to represent a "map" of hexagons, either diplomacy or game.	Hex Classes	

Launch Combat	Static Method	Implement Combat Phase	Moveable Unit, Army Combat Results Table, HUD-Controller, MovementCalculator	None
Lobby	Class	A server-side object that manages a grouping of client connection.	Tag, Flag, MessagePhoenix	
MainMenuController	Controller	Define and limit access to main menu	Game.java	
Map Rendering Classes	View	Renders the game map and things on it	MapView, Hexagon Classes, Hex Edge Classes	
MapView	View	A GUI widget to act as a container for the Map Rendering code	Hexagon Classes, Map Rendering Classes	
MessagePhoenix	Utility Class	Used to pack, unpack, send and receive messages over the network	Tag, Flag	
Moveable Unit	Class	Unit Super-Class	None	ArmyUnit, Character, Monster
MovementCalculator	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
OverStackWarning	Static Method	Notifies player of an overstack condition.	UnitPool	HUDController

RemoveOverStack	Method	Removes units from an overstacked hex.	UnitPool	HUDController
RemoveUnit	UnitPool Method	Removes a single unit.	Sorted map: pool	None
Retreat	Static Method	Implement Retreat after Combat	Launch Combat, MovementCalculator	None
SetSafeTeleport	UnitPool Method	Sets the units that can safely teleport.	Array list: safeTeleport	None
Solar Display	Class	Tracks solar position, updates HUD image	Scenario/Game Rules	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

4.4 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 preform correctly all of the needed variables need to be filled out.

4.5 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.

4.6 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

4.7 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.

4.8 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.

4.9 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.

4.10 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 puposes, the demands on the NetworkServer are fairly minimal, and few performance issues have arisen.

4.11 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.

4.12 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.

4.13 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.

4.14 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.

4.15 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.

4.16 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.

4.17 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 preform correctly all of the member variables need to be filled out.

4.18 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.

4.19 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.

4.19.1 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.

4.20 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.

4.21 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.

4.22 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.

4.23 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.

4.24 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.

4.25 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.

4.26 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.

4.27 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.

4.28 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.

4.29 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.

4.30 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.

4.31 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.

4.32 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.

4.33 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.

4.34 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.

4.35 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.

4.36 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.

4.37 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.

4.38 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.

4.39 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.

4.40 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.

4.41 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.

4.42 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.

4.43 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.

4.44 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
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

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 MoveableUnit

Requirements Description

Implementation Description

Differences

5.2 Traceability Analysis

[Describe the consistency of our requirements descriptions and implementation in general]

6 Appendix A