Nicolas C. Broeking & Josh Rahm
April 27, 2014
Project Part 4

Risk: Part 4

1.) Features

We implemented a command line version of risk. You can either run a risk server application or a risk game application. When you run a risk game application it attempts to connect to a risk server application. If it fails the game exits. If it makes a connection then the server attempts to join it with another connecting player. Once a connection is made the two players can begin playing a game of risk. The players whose turn it is may attack the opposing player. The game will continue to shift back and forth between players until the game is over. Once the game is over the application prints if you are a winner or a loser and then exits.

The class diagram is in appendix A.

2.) Design pattern

Singleton: We used the singleton design pattern for the Application classes and for the marshaling strategy classes.

Strategy: We used the strategy design pattern for the NotificationPool classes to decide how to marshal data from and to an IO stream.

Observer: We used the observer pattern to subscribe and unsubscribe observers to the notification pools as all of our IO was done asynchronously using threads.

Chain of Command: We used chain of command to find the correct marshaler for a chunk of data read from an input stream and the strategy that successfully decodes the data is the one that handles it.

Proxy: We used a proxy design pattern for our ServerProxy and ClientProxy. Anytime a process needs to communicate with either the server or the game then it would write or read from the proxy as a means of better abstracting the remote communication.

Prototype: We used prototype for the events to allow us to easily clone an event from a prototype when we read an event for processing.

State: The server application is a state machine to better abstract the behavior of the server and the game based on previous events.

3.) Compare and Contrast

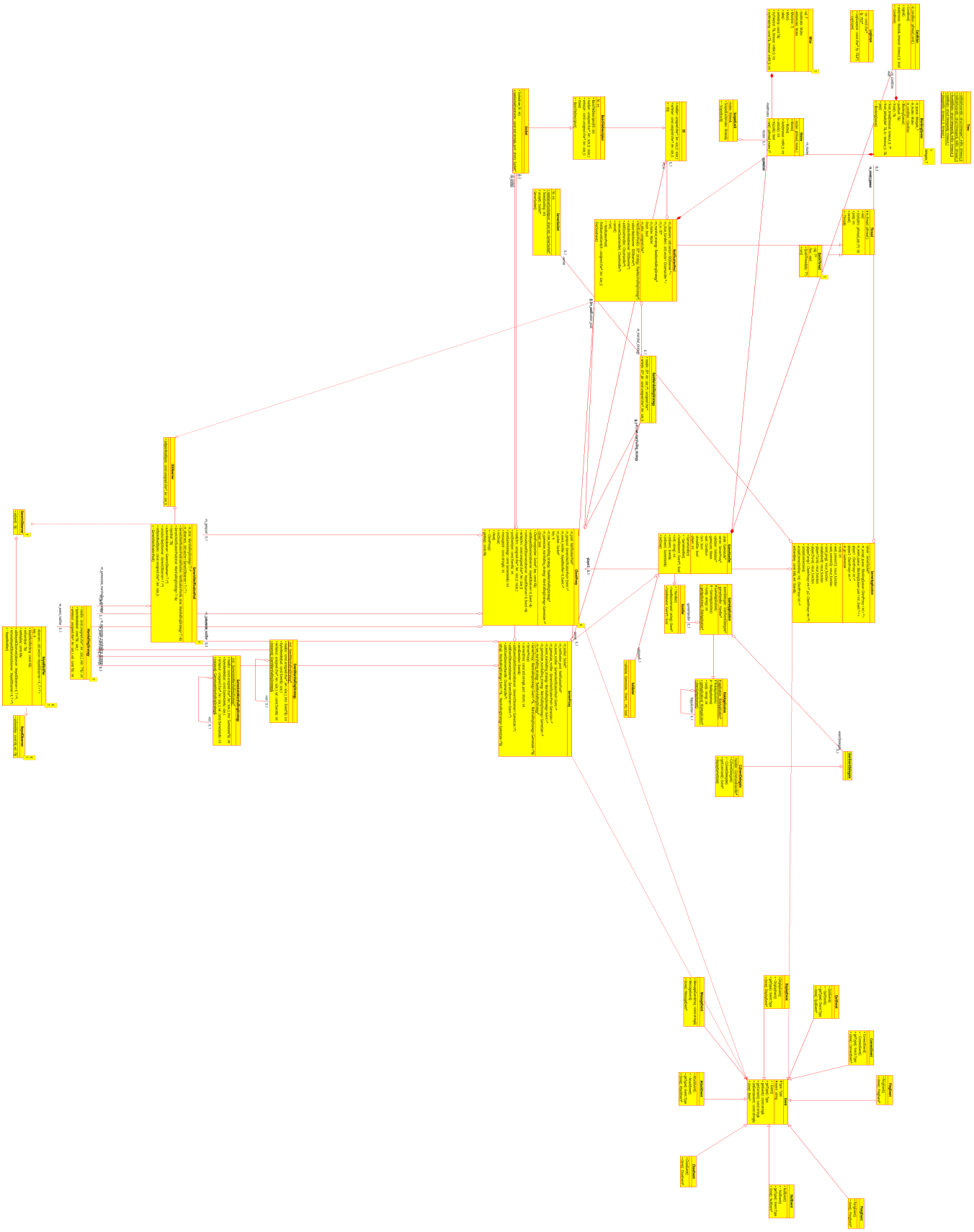
The class diagram that was done for project 2 only included the high level abstraction. We didn't realize how many more classes and patterns it would take to properly implement two machines talking to each other. So our current design includes many more classes regarding how the server and the client communicate. In addition to this we didn't end up using the AppDelegate class or a player class. The only thing that maintained the same is that the game application has UserEventDelegates that are used to control events.

The class diagram from part 2 is included in Appendix B.

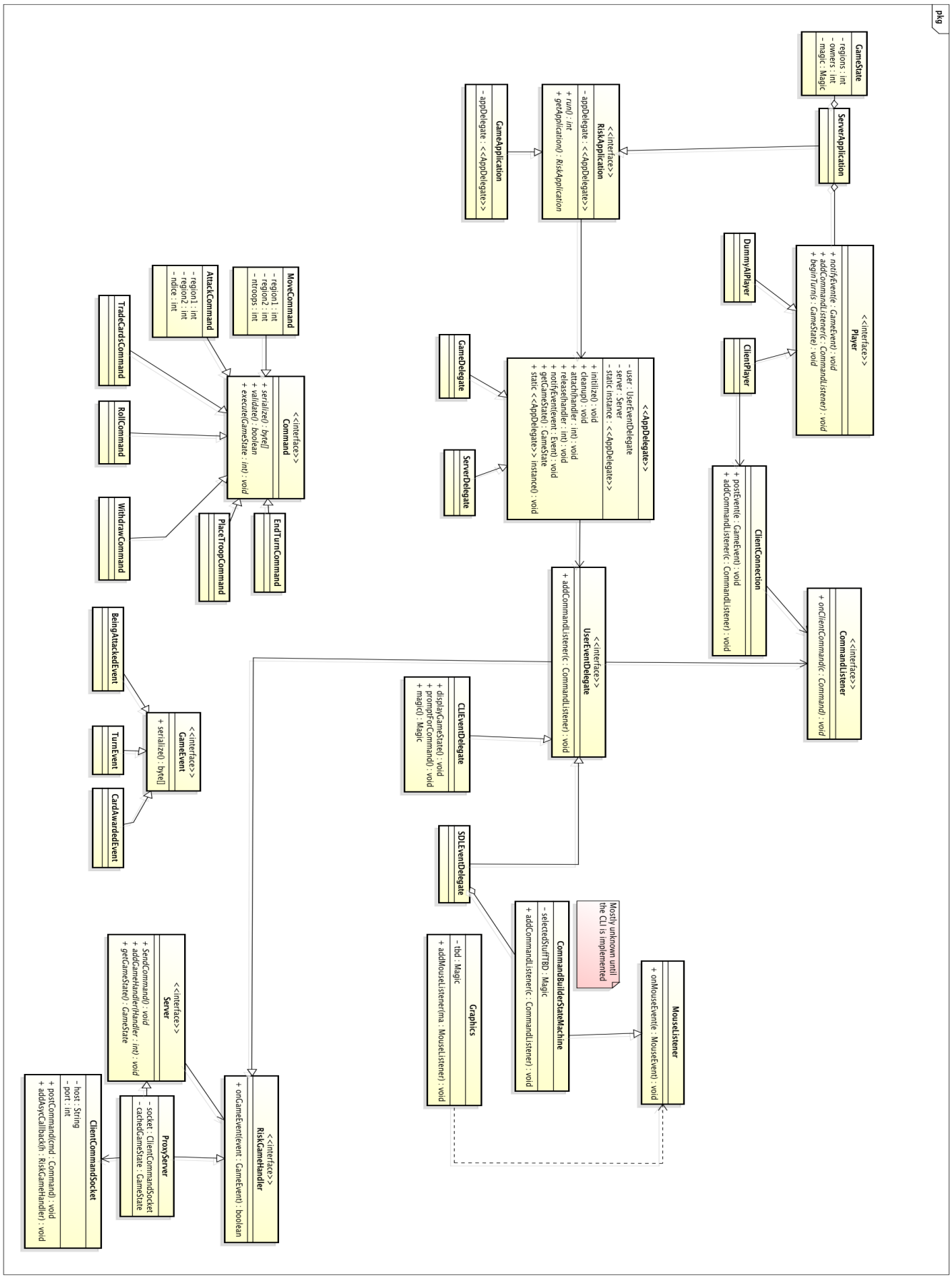
4.) What we learned

We learned that if you use design patterns to create boilerplate code before you start trying to make the application run then when you start putting the application together you make fewer errors and it is a much easier process. Design patterns are extremely beneficial to the implementation of larger projects.

Appendix A



Appendix B


pkg
GameState
- regions : int
- owners : int
- magic : Magic
ServerApplication
<<interface>>
Player
+ notifyEvent(e : GameEvent) : void
+ addCommandListener(c : CommandListener) : void
+ beginTurn(s : GameState) : void
DummyAIPlayer
ClientPlayer
<<interface>>
RiskApplication
- appDelegate : <<AppDelegate>>
+ run() : int
+ getApplication() : RiskApplication
GameApplication
- appDelegate : <<AppDelegate>>
<<AppDelegate>>
- user : UserEventDelegate
- server : Server
- static instance : <<AppDelegate>>
+ initilized() : void
+ cleanup() : void
+ attachHandler : (int) : void
+ releaseHandler : (int) : void
+ notifyEvent(event : Event) : void
+ getGameState() : GameState
+ static <<AppDelegate>> instance() : void
GameDelegate
ServerDelegate
ClientConnection
+ postEvent(e : GameEvent) : void
+ addCommandListener(c : CommandListener) : void
<<interface>>
CommandListener
+ onClientCommand(c : Command) : void
<<interface>>
UserEventDelegate
+ addCommandListener(c : CommandListener) : void
CLIEventDelegate
+ displayGameState() : void
+ promptForCommand() : void
+ magic() : Magic
SDLEventDelegate
Mostly unknown until the CLI is implemented
CommandBuilderStateMachine
- selectedStuffTBD : Magic
+ addCommandListener(c : CommandListener) : void
Graphics
- tbd : Magic
+ addMouseListener(ma : MouseListener) : void
MouseListener
+ onMouseEvent(e : MouseEvent) : void
<<interface>>
Command
+ serialized() : byte[]
+ validate() : boolean
+ executeGameState : int) : void
MoveCommand
- region1 : int
- region2 : int
- ntroops : int
AttackCommand
- region1 : int
- region2 : int
- ndice : int
TradeCardsCommand
RollCommand
WithdrawCommand
PlaceTroopCommand
EndTurnCommand
<<interface>>
GameEvent
+ serialized() : byte[]
BeingAttackedEvent
TurnEvent
CardAwardedEvent
<<interface>>
Server
+ SendCommand() : void
+ addGameHandlerHandler : (int) : void
+ getGameState() : GameState
ProxyServer
- socket : ClientCommandSocket
- cachedGameState : GameState
<<interface>>
RiskGameHandler
+ onGameEventevent : GameEvent() : boolean
ClientCommandSocket
- host : String
- port : int
+ postCommand(cmd : Command) : void
+ addAsyCallback(h : RiskGameHandler) : void