Tanks

McKinley Olsen

December 7, 2012

Contents

1 Functional Requirements

1. A player can register or unregister themselves with the fight manager at any time.
2. A player can move around in a physical space, as long as the following constraints are observed:
   (a) The rate (distance / time) at which a player can move is limited.
   (b) A player cannot move if they are trying to get a shell from the Shell Manager, or are filling a shell with the Gunpowder Manager.
   (c) A player that violates these constraints is removed from all fights and unregistered with the Fight Manager
3. A player can fire non-empty shells at any player.
   (a) The Fight Manager can broadcast a fired shell's location to all players. Every player at that location must reply back that they were hit.
   (b) If a player responds that they were hit, but took longer than 10ms to reply to the Fight Manager, the Fight Manager will remove that player from all games and de-register them.
   (c) A player that fires and hits another player that is not participating in any of the games that player is participating creates a new fight.
4. A player can ask the Fight Manager for a list of a players most recent known locations (up to 10).
   (a) The Fight Manager can request the last 10 locations from a player, and the player will respond with their last 10 locations.
   (b) If the response of a player with their locations takes longer than 10ms, the Fight Manager will remove that player from all games and de-register them.
5. A player can ask the Fight Manager for a list of current players.
   (a) Players can also ask the Fight Manager for a list of players in a specific fight.
6. A player can instigate a fight at any time, regardless of what other fights are in progress.
   (a) A player instigates a fight by telling the fight manager that it is launching a shell (filled with some percentage of gunpowder) at a player that is believed to be at a specified location.
   (b) If the location of the other player was correct, then the player is hit with the shell and the fight begins.
   (c) The instigator and the player hit with the shell are automatically enrolled in the new fight.
7. Players can join fights in progress at any time.

    (a) Players can request a list of in progress fights from the fight manager.

8. Players do not have to take turns when fighting; they are able to perform an action they choose at any time, so long as the action is in accordance with the aforementioned restrictions.

9. Players can request an empty shell from the shell manager, but only hold up to 4 shells at a time.

10. A player can fill an empty shell by making a request to the gun powder manager to fill the shell to a specified percentage.

11. Once a player has been hit with shell(s) that contained an accumulated 100 or more cups of gunpowder, the player is removed from all fights and de-registered.

12. A fight manager must be able to update a SOAP webservice with information about itself and the game

    (a) The manager must provide the following information when started, which should be easily specifiable before starting the manager:

        i. The URL of the webservice
        ii. A name for the manager
        iii. The name of the person operating the manager
        iv. The e-mail address of the person operating the manager
        v. A global unique identifier (GUID) to identify the manager

    (b) The manager must inform the webservice when a new fight is started, providing to it the id of the fight and the manager's GUID

    (c) The manager must update the webservice on the current state of the game periodically. This should include information such as:

        i. The time the update message was sent
        ii. The current status of the game
        iii. The current number of players
        iv. The maximum number of players that have played
        v. The number of shells that have been launched in the game
        vi. The amount of gunpowder that has been launched
        vii. The amount of shells that hit players
        viii. The amount of gunpowder that hit players
        ix. The name of the winning player, if the game has been won

2 Protocols

2.1 Introduction

This section describes the protocols required in order to communicate between processes in Virtual Tanks.

2.2 Overview

Detailed below are the protocols used in the Virtual Tanks and their important characteristics. The communication patterns utilized are as follows:

- Request-Reply: a two-way communication from initiator to receiver(s), and from receiver(s) to initiator. As a reply message is communicated by the receiver(s), this pattern is generally considered reliable.
- Custom: A non-standard sequence of messages is used for this protocol. The sequence of messages will be described later in the document.

Protocol	**Description**	**Initiator**	**Other Components Involved**	**Communication Pattern**
Register	Allows a player to register with the Fight Manager, making them visible to other players and managed by the Fight Manager. Initiated when a player wishes to begin playing the game.	Player	Fight Manager	Request-Reply
Unregister	Allows a player to unregister from all games and to have that player become unknown to the Fight Manager. This protocol is initiated when the player wishes to quit, when a general application error occurs, or initiated by the Fight Manager when the player has met or exceeded the number of cups of gunpowder it takes to lose (specified in requirement 13), or when a player takes more than 10ms to respond to a Shell Fired Request.	Player, Fight Manager	Fight Manager, Player	Request-Reply

Protocol	**Description**	**Initiator**	**Other Components Involved**	**Communication Pattern**
Fire Shell	Allows a player to fire a shell filled with gunpowder at a location. This request is sent to the Fight Manager, who then proceeds to broadcast the shell's location to all players, who in turn only respond if they were effected by the shell.	Player	Fight Manager, Players	Custom
Location List	Allows a player to be informed of the up to 10 most recent locations of another player. Initiated when the player wishes to know the whereabouts of another player.	Player	Fight Manager, Specified Player	Custom
Player List	Allows a player to either get a list of all the players currently registered, or all the players in a specified fight. Initiated when the player wishes to know about other players.	Player	Fight Manager	Custom
Create Fight	Allows a player to create a new fight. Initiated when a player fires upon and hits a player that is not currently participating in any of the fights the firing player is participating in.	Player	Fight Manager	Custom
Join Fight	Allows a player to enlist/delist to a fight. A player requesting to register for a fight they are not currently enlisted in joins the fight. Conversely, a player requesting to register for a fight they are currently enlisted in leaves the fight.	Player	Fight Manager	Request-Reply

Protocol	Description	Initiator	Other Components Involved	Communication Pattern
Fight List	Allows a client to be informed of the currently active fights. Initiated when the player wishes to see available fights.	Player	Fight Manager	Custom
Get Shell	Allows a player to request an empty shell from the Shell Manager. As specified in the requirements, a player may have up to 4 empty or non-empty shells at a time. Initiated when the player wishes to get a new shell.	Player	Shell Manager	Request-Reply
Fill Shell	Allows a player to fill an empty shell to a specified percentage. Initiated by the player when they wish to prepare a shell for firing.	Player	Gunpowder Manager	Request-Reply

3 Message Types and their Definitions

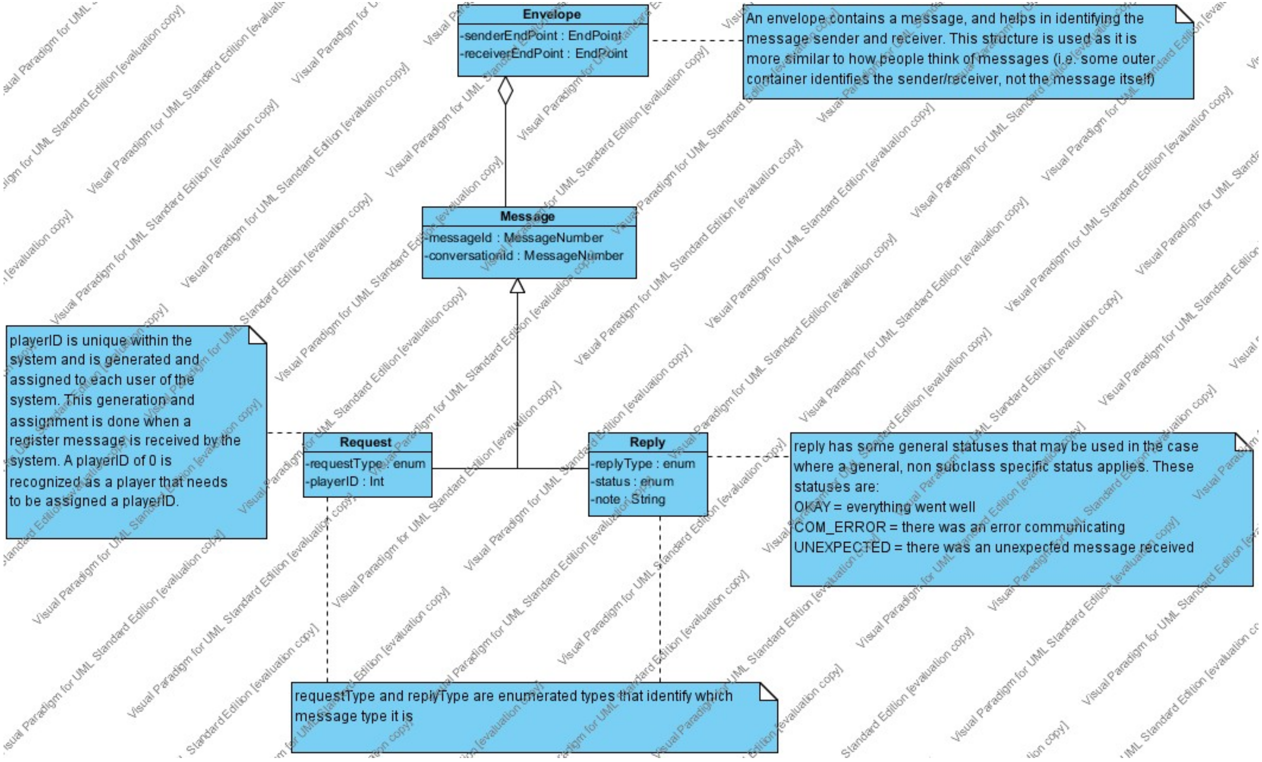
3.1 Introduction

Described here is the structure of the messages required to implement the aforementioned protocols. The structure is conveyed through class diagrams. These class diagrams omit certain methods, such as the constructor, getters/setters, create, encode, and decode. These methods are implied in the following diagrams, as their definition is relatively unimportant.

3.2 Diagrams

3.2.1 General Message Classes

Diagrams/Message Classes.jpg


Envelope
-senderEndPoint : EndPoint
-receiverEndPoint : EndPoint
Message
-messageId : MessageNumber
-conversationId : MessageNumber
Request
-requestType : enum
-playerID : Int
Reply
-replyType : enum
-status : enum
-note : String

An envelope contains a message, and helps in identifying the message sender and receiver. This structure is used as it is more similar to how people think of messages (i.e. some outer container identifies the sender/receiver, not the message itself)

playerID is unique within the system and is generated and assigned to each user of the system. This generation and assignment is done when a register message is received by the system. A playerID of 0 is recognized as a player that needs to be assigned a playerID.

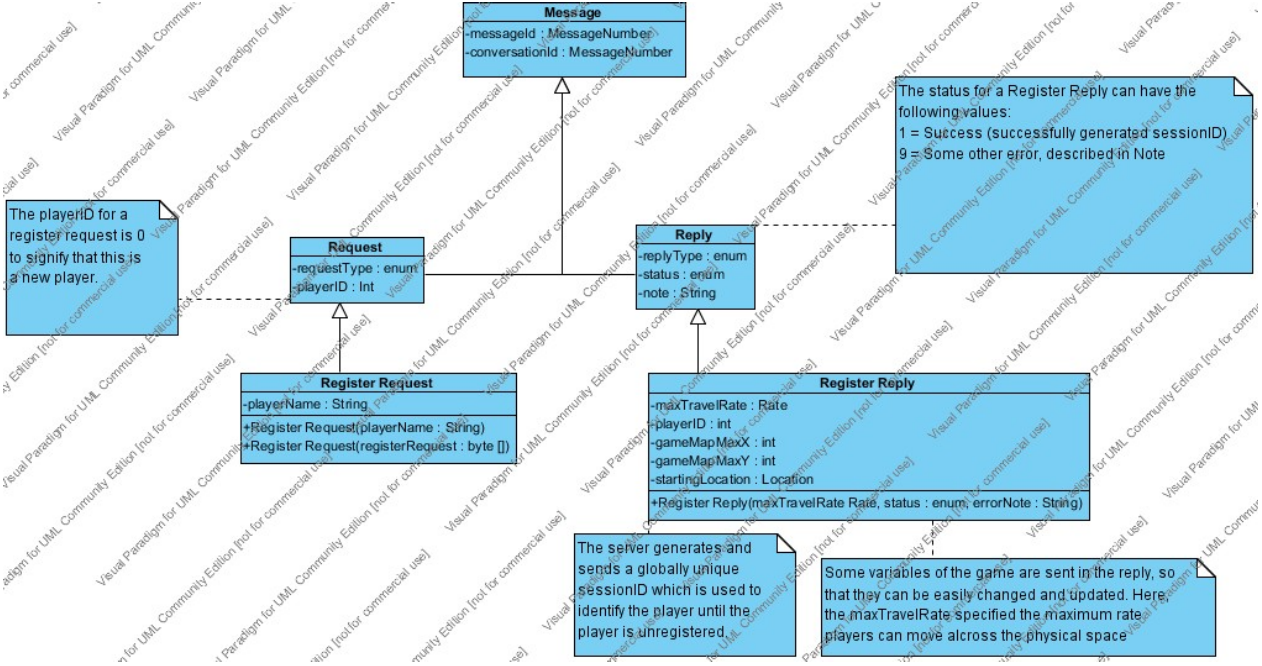
reply has some general statuses that may be used in the case where a general, non subclass specific status applies. These statuses are:
OKAY = everything went well
COM_ERROR = there was an error communicating
UNEXPECTED = there was an unexpected message received

requestType and replyType are enumerated types that identify which message type it is

Figure 1: The general superclasses used. All remaining classes inherit from these.

3.2.2 Register Protocol

Diagrams/Register Protocol.jpg


Message
-messageId : MessageNumber
-conversationId : MessageNumber
Request
-requestType : enum
-playerID : Int
Reply
-replyType : enum
-status : enum
-note : String
Register Request
-playerName : String
+Register Request(playerName : String)
+Register Request(registerRequest : byte [])
Register Reply
-maxTravelRate : Rate
-playerID : int
-gameMapMaxX : int
-gameMapMaxY : int
-startingLocation : Location
+Register Reply(maxTravelRate Rate, status : enum, errorNote : String)

The playerID for a register request is 0 to signify that this is a new player.

The status for a Register Reply can have the following values:
1 = Success (successfully generated sessionID)
9 = Some other error, described in Note

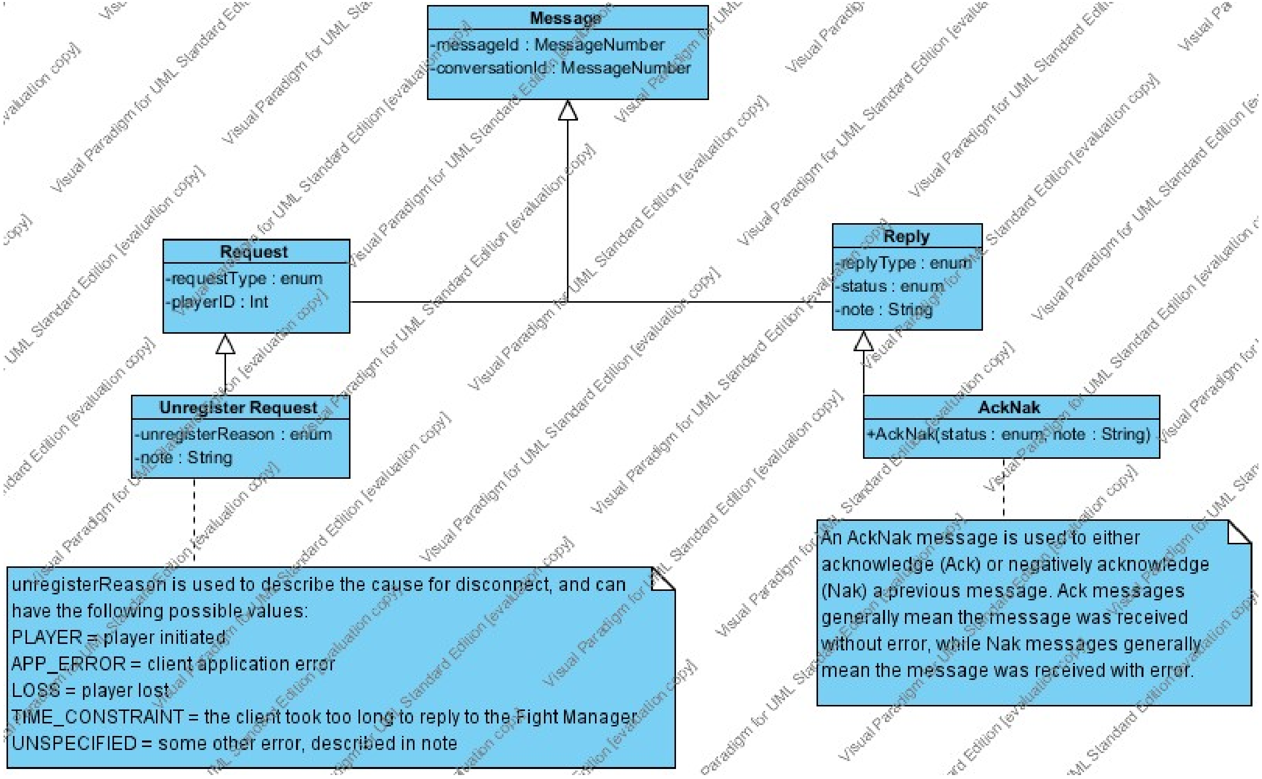
The server generates and sends a globally unique sessionID which is used to identify the player until the player is unregistered

Some variables of the game are sent in the reply, so that they can be easily changed and updated. Here, the maxTravelRate specified the maximum rate players can move alcross the physical space

Figure 2: The message classes used in the Register Protocol

3.2.3 Unregister Protocol

Diagrams/Unregister Protocol.jpg



unregisterReason is used to describe the cause for disconnect, and can have the following possible values:
PLAYER = player initiated
APP_ERROR = client application error
LOSS = player lost
TIME_CONSTRAINT = the client took too long to reply to the Fight Manager
UNSPECIFIED = some other error, described in note

An AckNak message is used to either acknowledge (Ack) or negatively acknowledge (Nak) a previous message. Ack messages generally mean the message was received without error, while Nak messages generally mean the message was received with error.

Figure 3: The message classes used in the Unregister Protocol

3.2.4 Location List Protocol

Diagrams/Location List Protocol.jpg

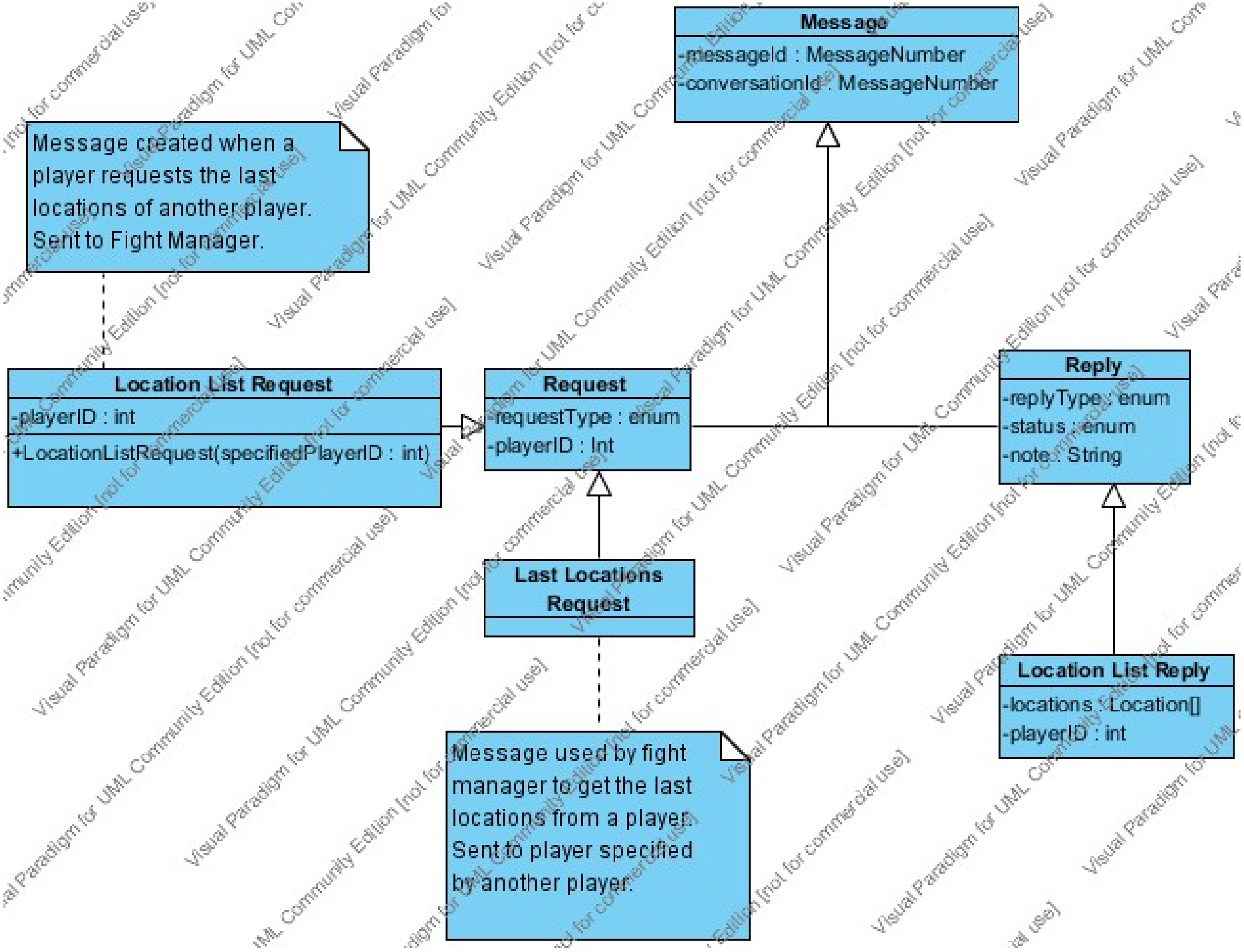

Message
-messageId : MessageNumber
-conversationId : MessageNumber
Message created when a player requests the last locations of another player. Sent to Fight Manager.
Location List Request
-playerID : int
+LocationListRequest(specifiedPlayerID : int)
Request
-requestType : enum
-playerID : Int
Reply
-replyType : enum
-status : enum
-note : String
Last Locations Request
Location List Reply
-locations : Location[]
-playerID : int
Message used by fight manager to get the last locations from a player. Sent to player specified by another player.

Figure 4: The classes used in the Location List Protocol

3.2.5 Player List Protocol

Diagrams/Player List Protocol.jpg

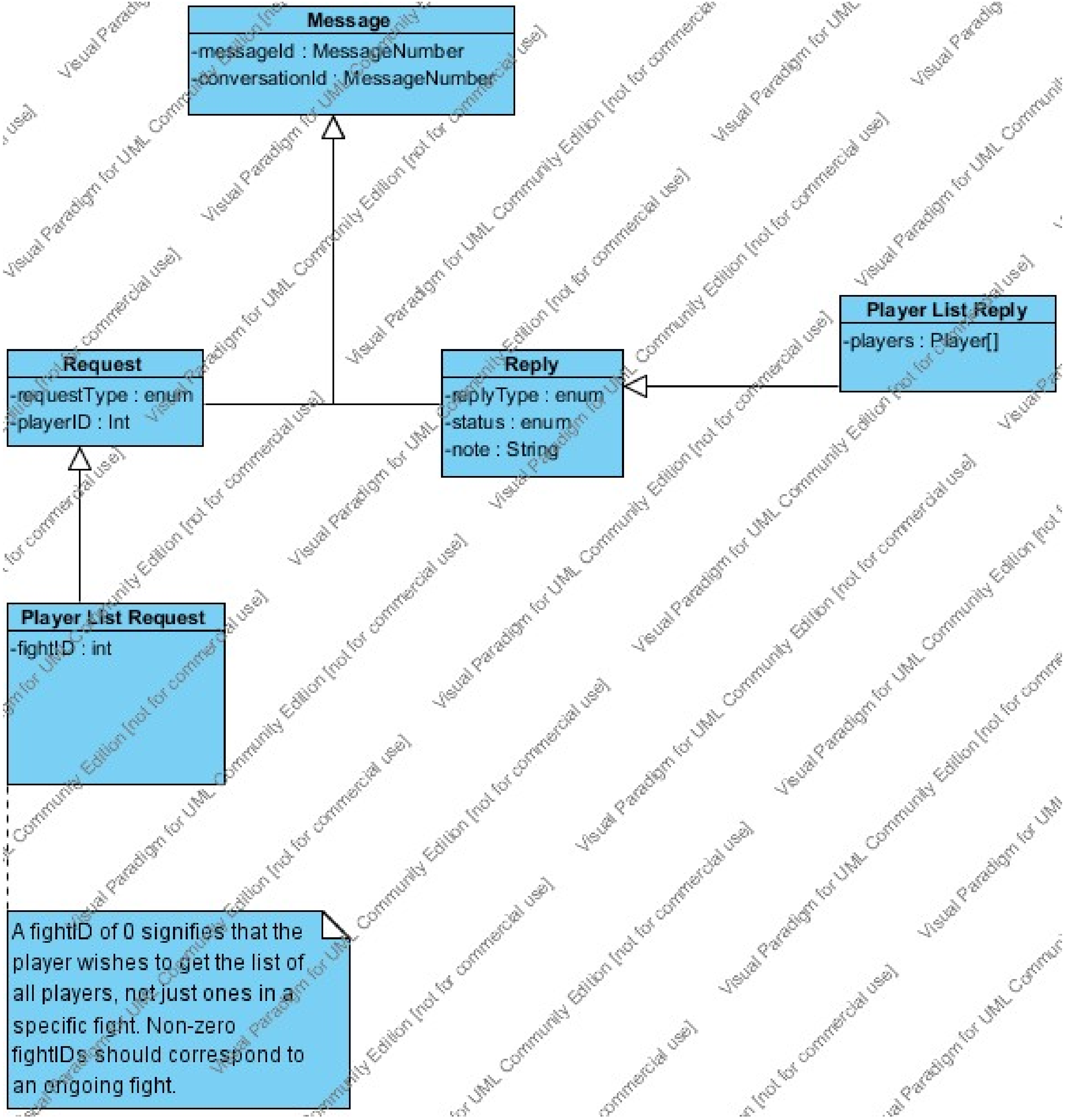

Message
-messageId : MessageNumber
-conversationId : MessageNumber
Request
-requestType : enum
-playerID : Int
Reply
-replyType : enum
-status : enum
-note : String
Player List Reply
-players : Player[]
Player List Request
-fightID : int
A fightID of 0 signifies that the player wishes to get the list of all players, not just ones in a specific fight. Non-zero fightIDs should correspond to an ongoing fight.
Visual Paradigm for UML Community Edition [not for commercial use]

Figure 5: The classes used in the Player List Protocol

3.2.6 Fight List Protocol

Diagrams/Fight List Protocol.jpg

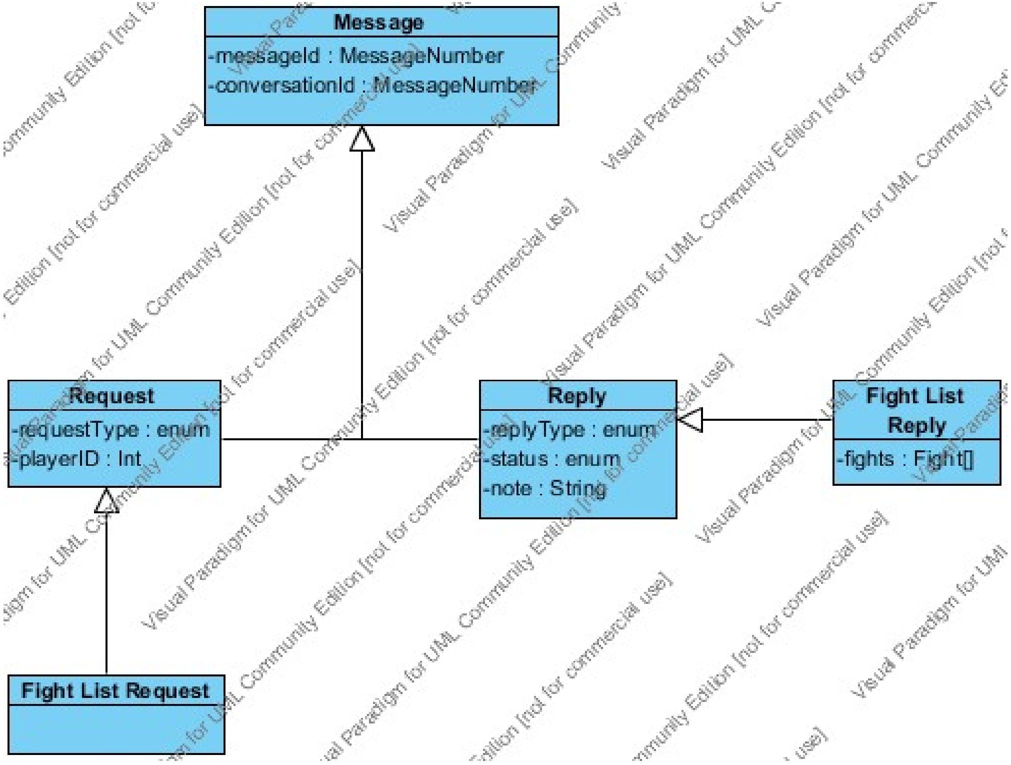


Figure 6: The classes used in the Fight List Protocol

3.2.7 Join Fight Protocol

Diagrams/Join Fight Protocol.jpg

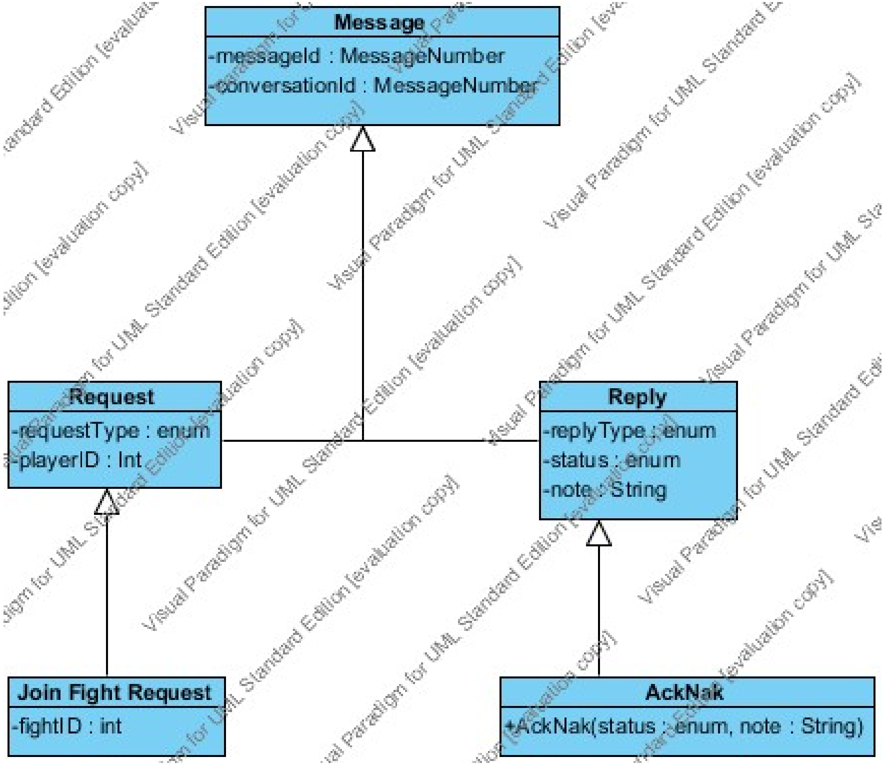

Message
-messageId : MessageNumber
-conversationId : MessageNumber
Request
-requestType : enum
-playerID : Int
Reply
-replyType : enum
-status : enum
-note : String
Join Fight Request
-fightID : int
AckNak
+AckNak(status : enum, note : String)
Visual Paradigm for UML Standard Edition [evaluation copy]

Figure 7: The classes used in the Join Fight Protocol

3.2.8 Create Fight Protocol

Diagrams/Create Fight Protocol.jpg

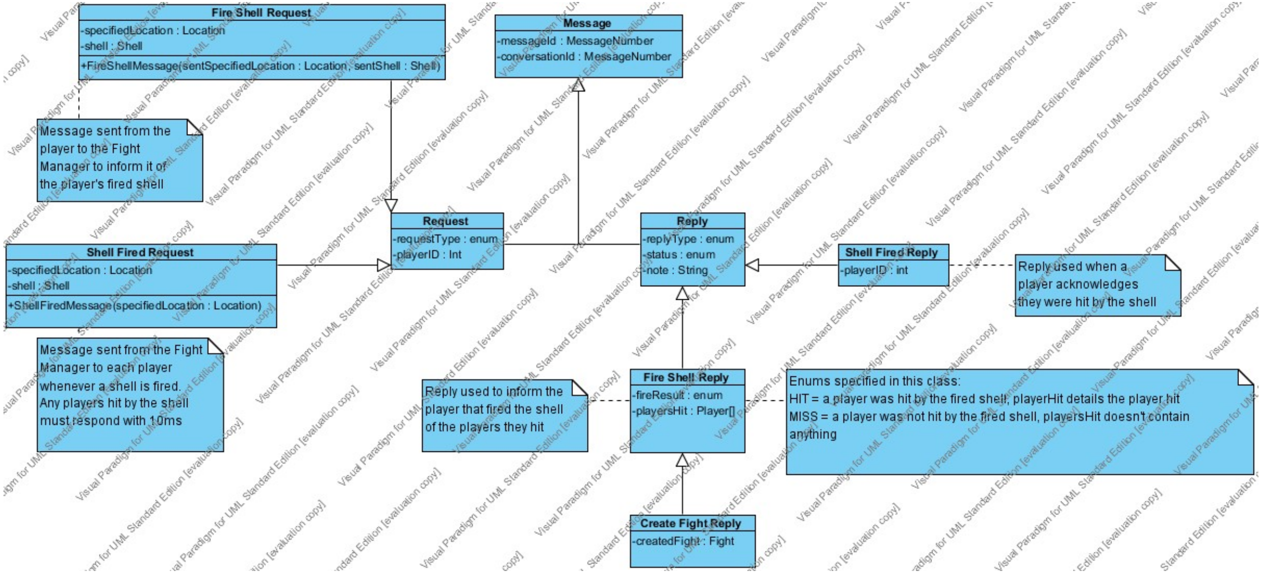

Fire Shell Request
-specifiedLocation : Location
-shell : Shell
+FireShellMessage(sentSpecifiedLocation : Location, sentShell : Shell)
Message
-messageId : MessageNumber
-conversationId : MessageNumber
Message sent from the player to the Fight Manager to inform it of the player's fired shell
Request
-requestType : enum
-playerID : Int
Reply
-replyType : enum
-status : enum
-note : String
Shell Fired Request
-specifiedLocation : Location
-shell : Shell
+ShellFiredMessage(specifiedLocation : Location)
Shell Fired Reply
-playerID : int
Reply used when a player acknowledges they were hit by the shell
Message sent from the Fight Manager to each player whenever a shell is fired. Any players hit by the shell must respond with 10ms
Reply used to inform the player that fired the shell of the players they hit
Fire Shell Reply
-fireResult : enum
-playersHit : Player[]
Enums specified in this class:
HIT = a player was hit by the fired shell; playerHit details the player hit
MISS = a player was not hit by the fired shell, playersHit doesn't contain anything
Create Fight Reply
-createdFight : Fight

Figure 8: The classes used in the Create Fight Protocol

3.2.9 Fire Shell Protocol

Diagrams/Fire Shell Protocol.jpg

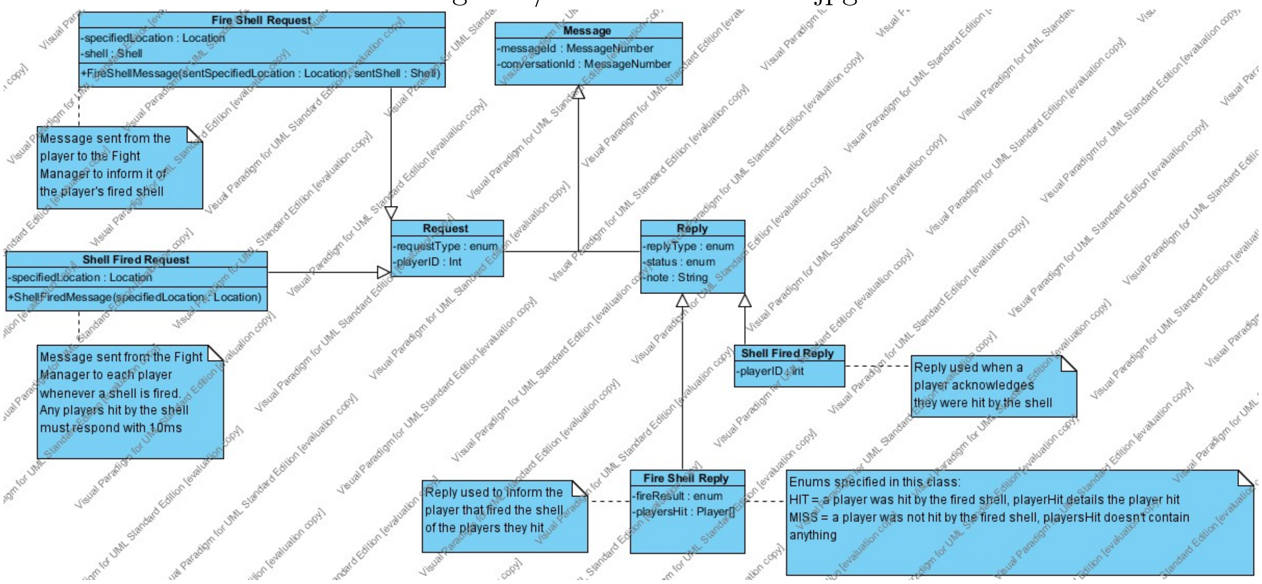

Fire Shell Request
-specifiedLocation : Location
-shell : Shell
+FireShellMessage(sentSpecifiedLocation : Location, sentShell : Shell)
Message
-messageId : MessageNumber
-conversationId : MessageNumber
Message sent from the player to the Fight Manager to inform it of the player's fired shell
Request
-requestType : enum
-playerID : Int
Reply
-replyType : enum
-status : enum
-note : String
Shell Fired Request
-specifiedLocation : Location
+ShellFiredMessage(specifiedLocation : Location)
Message sent from the Fight Manager to each player whenever a shell is fired. Any players hit by the shell must respond with 10ms
Shell Fired Reply
-playerID : int
Reply used when a player acknowledges they were hit by the shell
Reply used to inform the player that fired the shell of the players they hit
Fire Shell Reply
-fireResult : enum
-playersHit : Player[]
Enums specified in this class:
HIT = a player was hit by the fired shell, playerHit details the player hit
MISS = a player was not hit by the fired shell, playersHit doesn't contain anything

Figure 9: The classes used in the Fire Shell Protocol

3.2.10 Fill Shell Protocol

Diagrams/Fill Shell Protocol.jpg

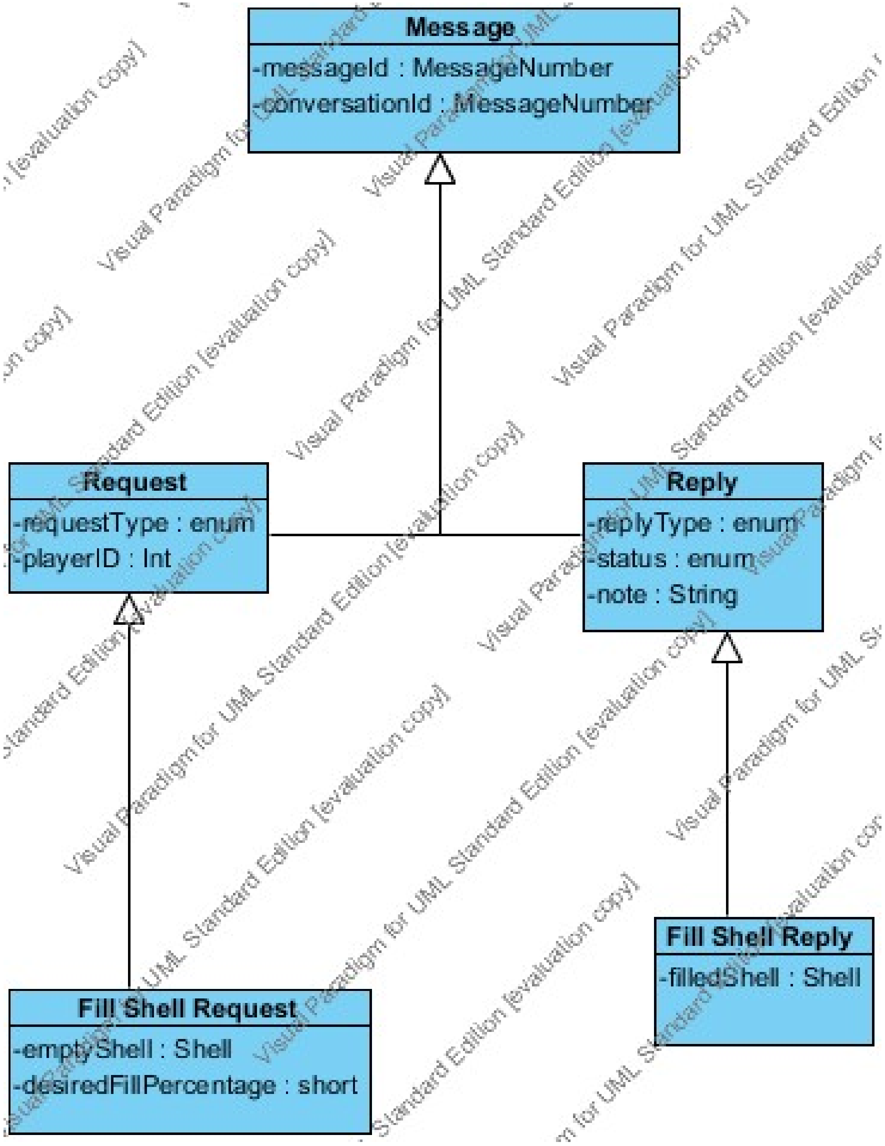


Figure 10: The classes used in the Fill Shell Protocol

3.2.11 Get Shell Protocol

Diagrams/Get Shell Protocol.jpg

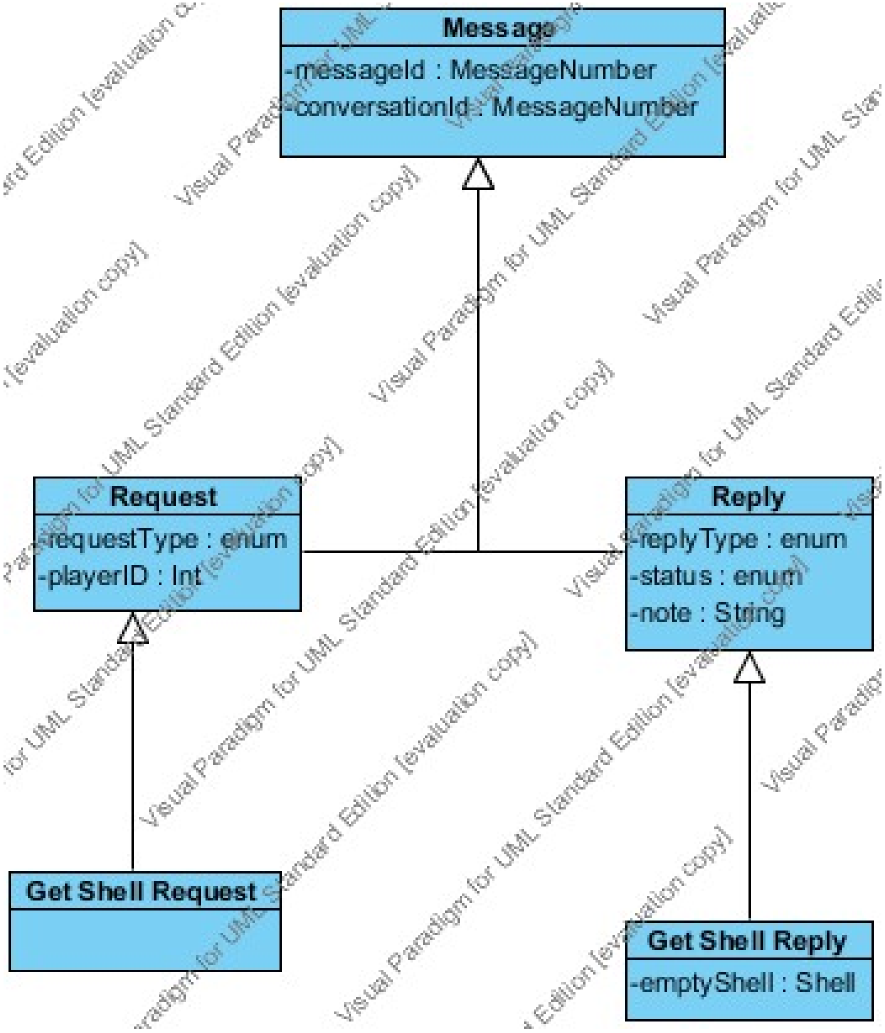


Figure 11: The classes used in the Get Shell Protocol

3.3 Message Descriptions

Note: A value in the Valid Attribute Values section that begins with an uppercase letter is a custom object used in the game that, as of yet, has not been fully specified.

Name	**Description**	**Uses**	**Valid Attribute Values**
Register Request	Intended to be used when a client needs to register with the Fight Manager. This message is used to notify the Fight Manager of a new player that must be assigned a player ID.	Register Protocol	playerName: ”John Smith”
Register Reply	Sent in reply to a Register Request, a client receiving this message knows they have been registered with the Fight Manager, assigned a unique player ID, and are available to be fired upon.	Register Protocol	playerID: 1 rate: Rate gameMapMaxX: 100 gameMapMaxY: 100 startingLocation: Location
Unregister Request	Means that a player will be unregistered with the Fight Manager, and that their communication will break. Used when the player quits the game, used by the Fight Manager to inform a player of their forced disconnect.	Unregister protocol	unregisterReason: UNSPECIFIED note:“System Restart”
Fire Shell Reply	Used to inform a player of the effect of their shell fire. Means the shell fire of the player was received successfully and that its effects have been noted.	Fire Shell Protocol	fireResult: HIT

Name	**Description**	**Uses**	**Valid Attribute Values**
			playersHit: Player[]
Location List Request	Used when a player instigates a request for a another player's last locations. Means the Fight Manager must query the specified player and find out their last locations.	Location List Protocol	specifiedPlayerID: 1
Last Locations Request	Used when the Fight Manager needs to ask a player their last locations. Means the player must know and gather their last locations to send back.	Location List Protocol	N/A
Location List Reply	Used when a player responds to a Last Locations Request and sends back their previous locations. Means the Fight Manager has the information the player requested with the Location List Request, and can send this message onto the player that requested the information.	Location List Protocol	locationList: Location[]
Player List Request	Used when a player requests a list of current players. Specifying a fight ID of 0 denotes a request for a list of all players, specifying a valid fight ID denotes a request for a list of players in that fight. Means the Fight Manager must put together a list of either all players or players in a specific fight and send that list back.	Player List Protocol	fightID: 0

Name	**Description**	**Uses**	**Valid Attribute Values**
Player List Reply	Used when the Fight Manager needs to send a list of players to the player that requested the list. This message is special, in that it is sent over a TCP connection created to transfer it. To the player, the receipt of this message means the created TCP connection may now be closed.	Player List Protocol	players: Player[]
Fight Creation Reply	Used when a player fired a shell at a location that effected another player that was not in any of the fights they were participating. Means a new game has been created that the player is now enrolled in.	Create Fight Protocol	createdFight: Fight
Join Fight Request	Used when a player wishes to join an in-progress fight. Means the player has specified a fight to join by a fight ID, and that the player needs to be enrolled in that fight.	Join Fight Protocol	fightID: 1
Fight List Request	Used when a player requests a list of in-progress fights. Means the Fight Manager must put together a list of all fights to return to the requesting player.	Fight List Protocol	N/A
Fight List Reply	Used when the Fight Manager needs to send a list of fights to the player that requested the list. This message is special, in that it is sent over a TCP connection created to transfer it. To the player, the receipt of this message means the created TCP connection may now be closed.	Fight List Protocol	fights: Fight[]

Name	**Description**	**Uses**	**Valid Attribute Values**
Get Shell Request	Used when a player requests a new shell and they do not currently have the maximum number possible. Means the Shell Manager must create a shell to return to the requesting player.	Get Shell Protocol	N/A
Get Shell Reply	Used when the Shell Manager returns a new shell to a player that requested one. Means the player has a new, empty shell of some capacity.	Get Shell Protocol	emptyShell: Shell
Fill Shell Request	Used when a player requests that an empty shell they received in the Get Shell Protocol be filled with gunpowder. Means the Gunpowder Manager must fill the shell some specified percentage full.	Fill Shell Protocol	emptyShell: Shell fillPercentage: 100
Fill Shell Reply	Used when the Gunpowder Manager has filled a previously received, empty shell, and is now ready to send this filled shell back to the requester. Means the player that requested the shell to be filled has received a filled shell with the specified percentage full.	Fill Shell Protocol	filledShell: Shell

Name	**Description**	**Uses**	**Valid Attribute Values**
Fire Shell Request	Used when a player fires a filled shell at another player that may or may not be participating in a fight they currently are participating in. Means the Fight Manager must broadcast this shell's firing to all players so that they may react to it.	Fire Shell Protocol, Create Fight Protocol	specifiedLocation: Location shell: Shell
Shell Fired Request	Used when the Fight Manager wishes to broadcast to all players the event that a shell has been fired. Means players must determine whether they were hit by the shell.	Fire Shell Protocol, Create Fight Protocol	specifiedLocation: Location
Shell Fired Reply	Used when a player determines a fired shell hit them. Means the Fight Manager must add this player to the list of players hit by the shell.	Fire Shell Protocol, Create Fight Protocol	playerID: 1
AckNak	Used when some component of the system wishes to either acknowledge or negatively acknowledge the correct receive of a message. Means a message previously sent by the receiving component reached the destination, and that this component must determine whether that message's transmittal succeeded.	Unregister Protocol, Join Fight Protocol	N/A

Name	Description	Uses	Valid Attribute Values

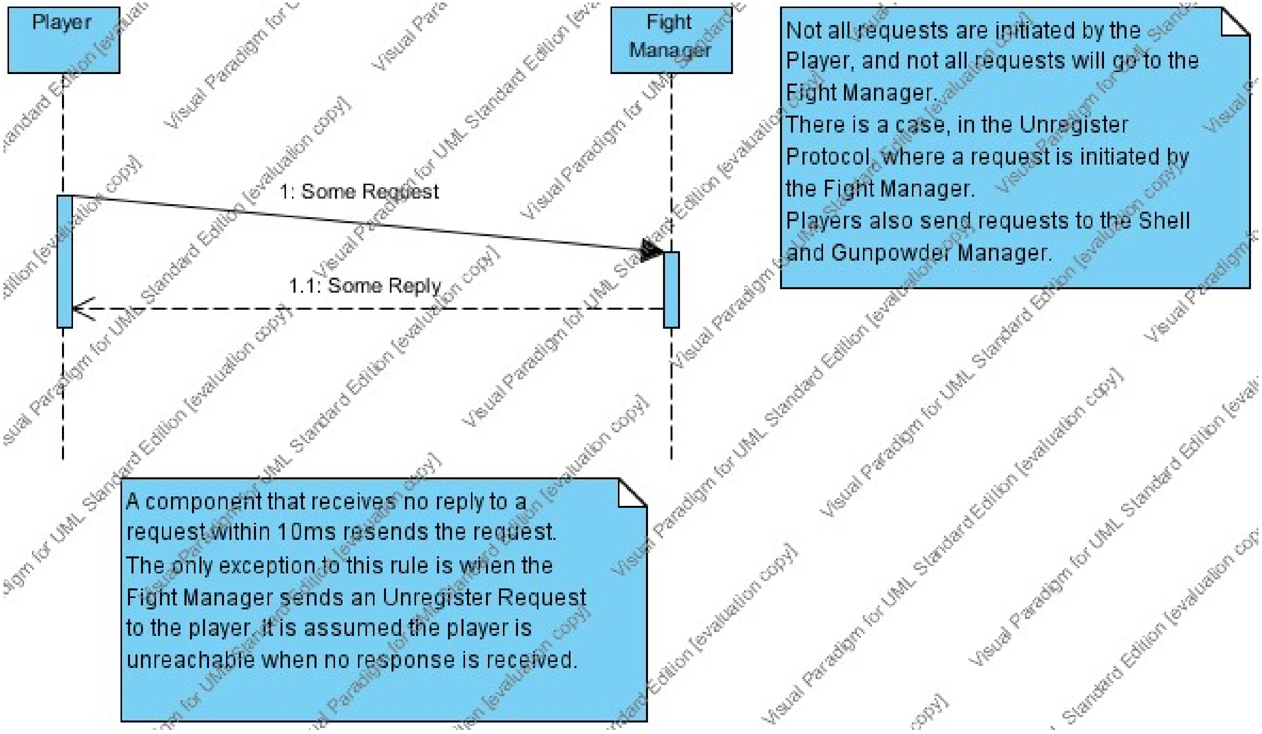
4 Message Sequences and Timing

4.1 Introduction

Described here is the sequence of messages and their timing in the aforementioned protocols. While the sequencing for each protocol utilizing a custom communication pattern is diagrammed here, a general, multi-use diagram is presented for all utilizing the familiar request-reply pattern.

4.2 General Request-Reply

Diagrams/General.jpg



Not all requests are initiated by the Player, and not all requests will go to the Fight Manager.
There is a case, in the Unregister Protocol, where a request is initiated by the Fight Manager.
Players also send requests to the Shell and Gunpowder Manager.

A component that receives no reply to a request within 10ms resends the request.
The only exception to this rule is when the Fight Manager sends an Unregister Request to the player. It is assumed the player is unreachable when no response is received.

Figure 12: Sequence and timing of messages in a general request-reply pattern

4.3 Location List Protocol

Diagrams/Location List Protocol Sequence.jpg

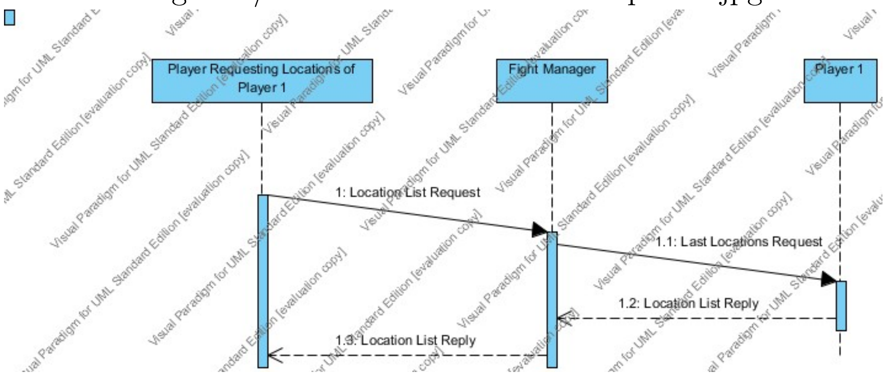


Figure 13: Sequence and timing of messages utilized in the Location List Protocol

4.4 Fire Shell Protocol

Diagrams/Fire Shell Protocol Sequence.jpg

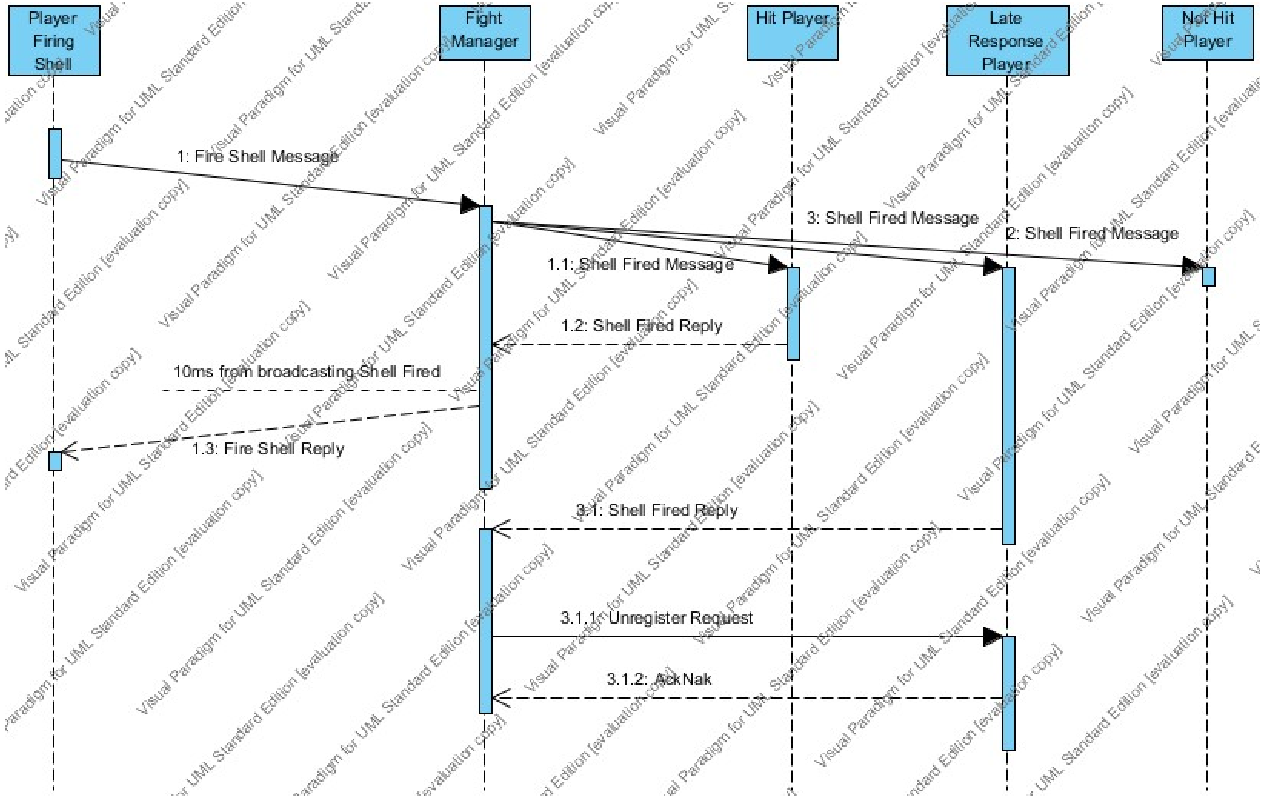


Figure 14: Sequence and timing of messages utilized in the Fire Shell Protocol

4.5 Create Fight Protocol

Diagrams/Create Fight Protocol Sequence.jpg

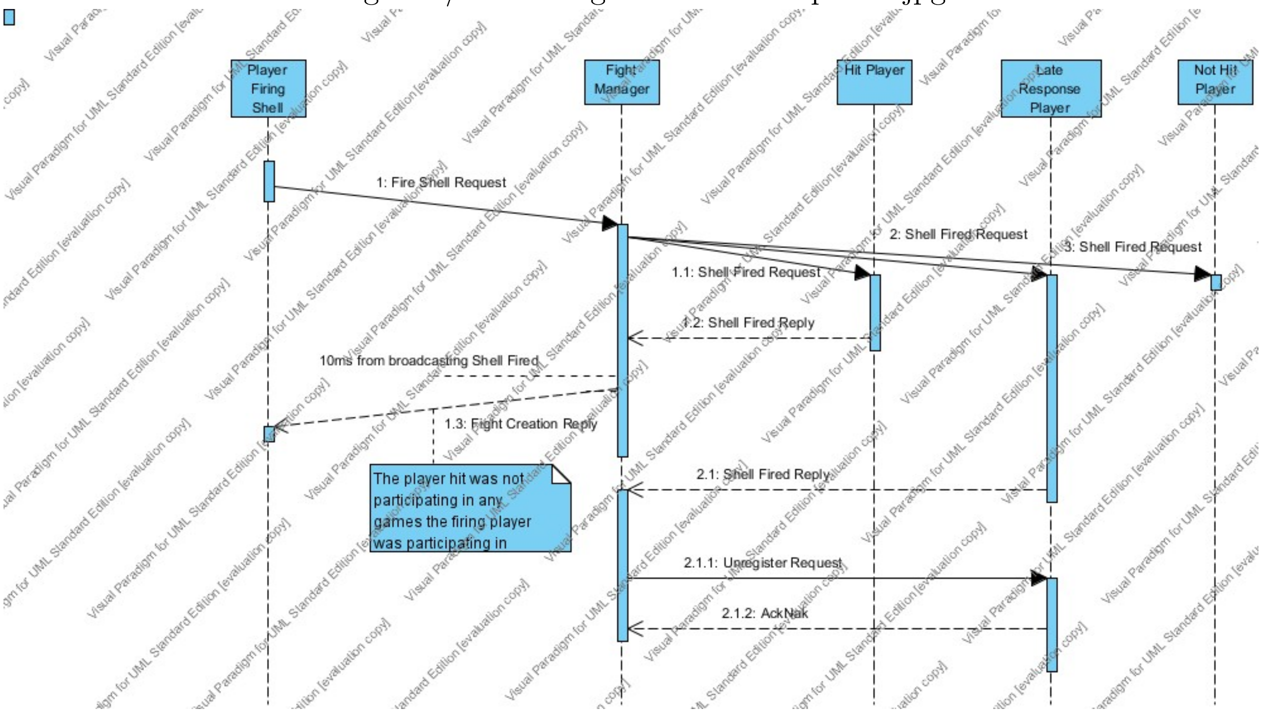


Figure 15: Sequence and timing of messages utilized in the Create Fight Protocol

5 Structure

5.1 Introduction

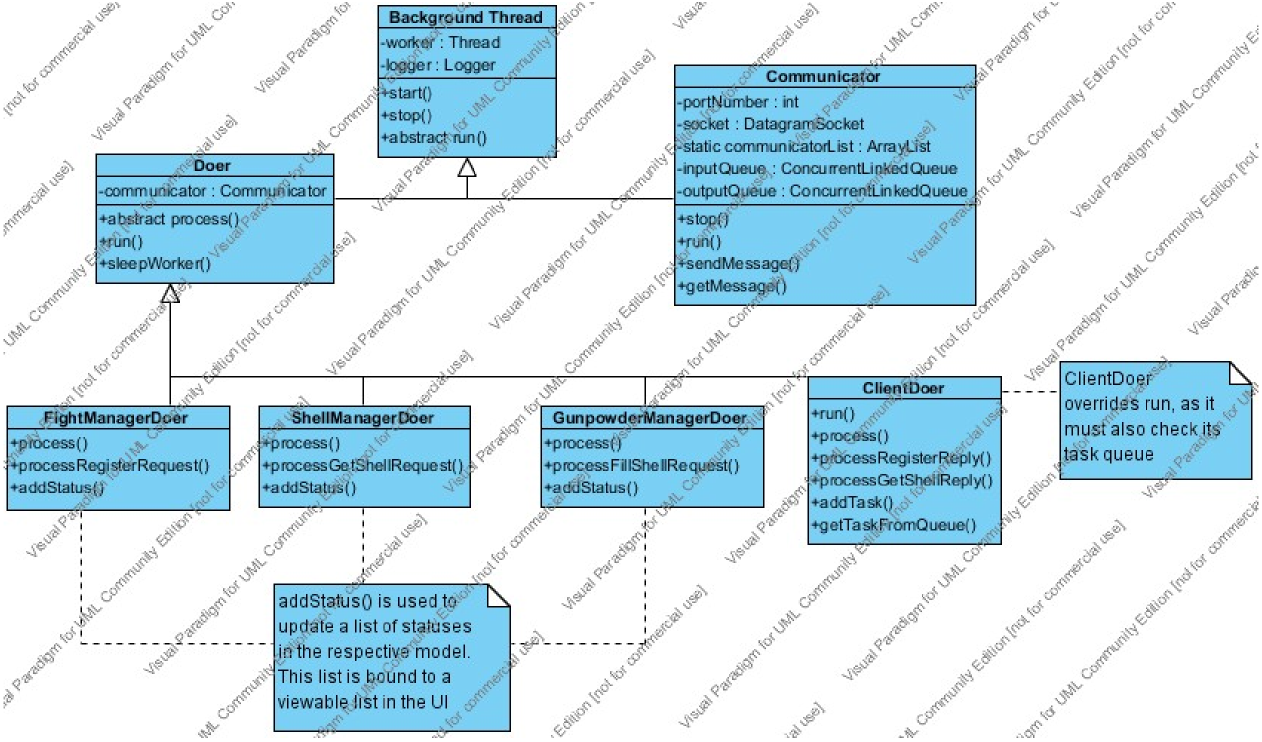
Described here is the structure of the needed components in the system, including the:

- Fight Manager
- Client
- Shell Manager
- Gunpowder Manager

5.2 Classes

5.2.1 Worker Classes

Diagrams/ThreadingStructure.jpg

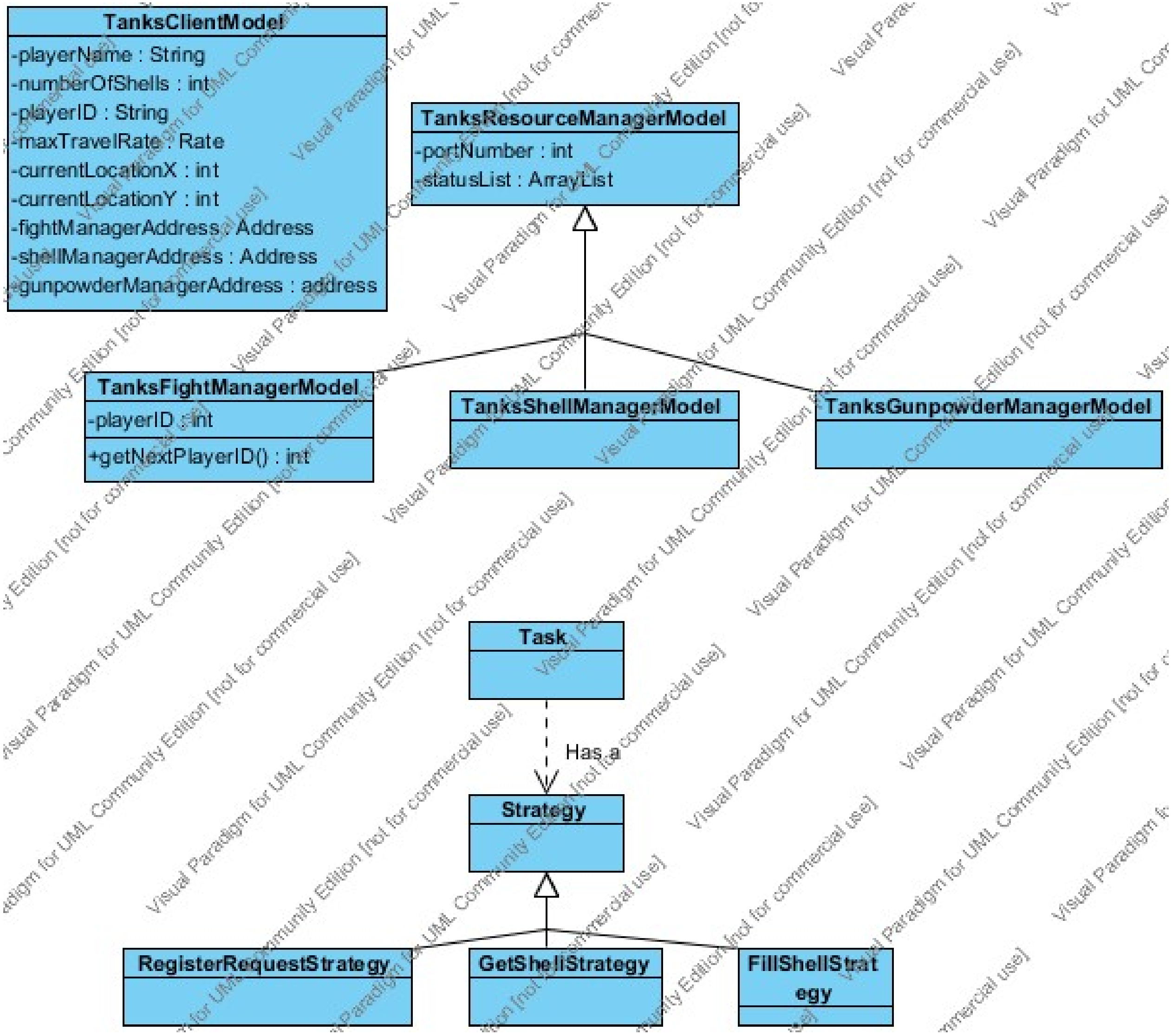


Described above are the classes in the system that contain threads that run independent of the main application thread. Each of these classes inherit from BackgroundThread, which describes basic operations inherent to all classes (i.e. start, stop, run). Communicators have a socket that their thread continually tries to receive from. Any received messages are put into an inputqueue, that a Doer can then access. A Doer has a Communicator, and continually checks its inputqueue for new messages. Any messages found are processed by a call to process, and the appropriate action is taken, depending on the type of Doer (FightManager, ShellManager, GunpowderManager, or Client).

All Manager Doers have an addStatus method that updates a list of statuses in the appropriate model. A UI list element is bound to this list, meaning anything added to the list in the model will also appear in the list in the UI.

5.2.2 Models and Strategies

Diagrams/ModelAndStrategyStructure.jpg



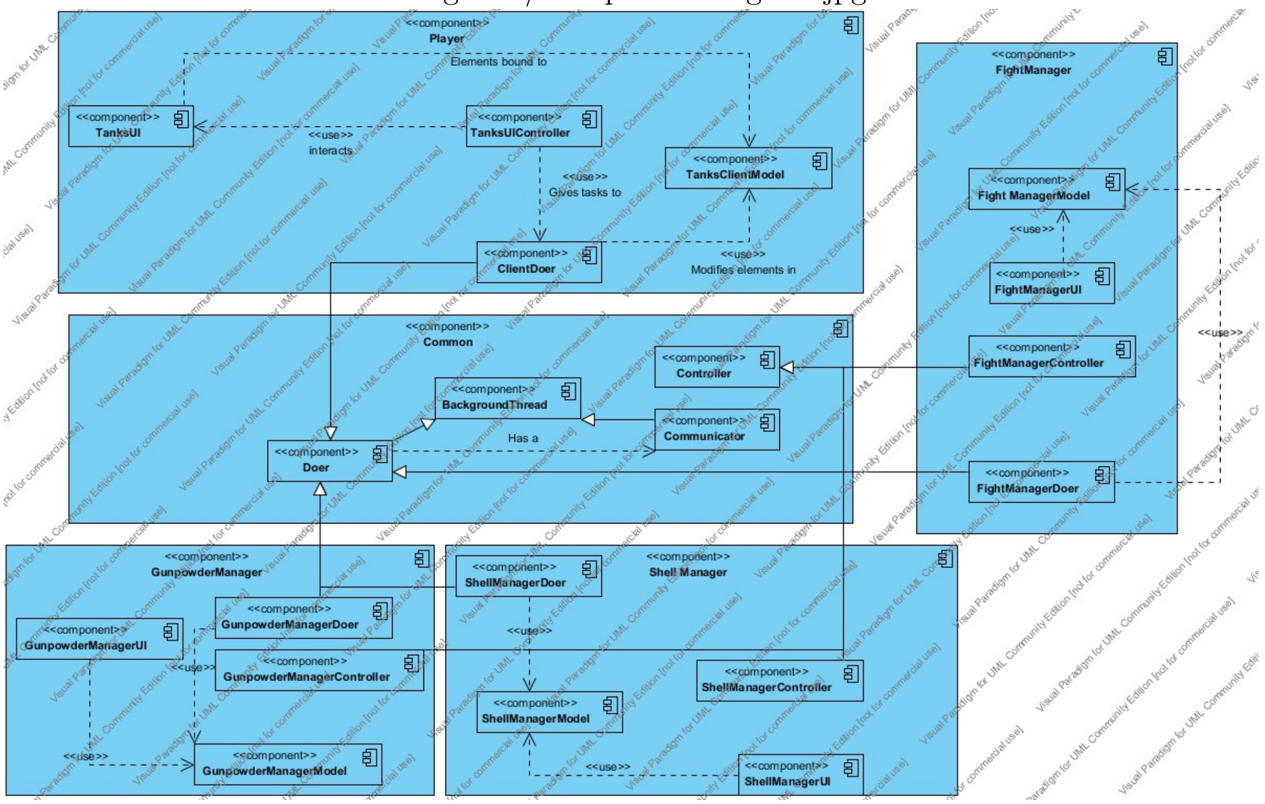
The above diagram depicts both the strategies and models used in the system.

A model is used to house data about the current state of the application. Things like registered players, player name, player ID, etc. are housed here. In the case of the Managers, the model is only changed and used by their respective Doer. The model used on the client side is used and changed by both the Doer and the Controller. This is done because the UI on the client side is interactive, and the model must change to reflect user choices and entries.

Strategies are only used on the client, and are kept within a task. The base class Strategy is abstract, and should not be instantiated. A task is used on the client application when the user has made some action that requires processing by the Doer (this processing usually includes sending message(s) to a manager(s). When this action occurs, a new task is created, along with the correct strategy (i.e. registration, get shell). The Strategy is added to the task, and the task put in the Doer's work queue. The Doer will check this task, and execute the strategy contained within the class.

5.3 Components

Diagrams/ComponentDiagram.jpg



<<component>> Player
Elements bound to
<<component>> TanksUI
<<use>> interacts
<<component>> TanksUIController
<<use>> Gives tasks to
<<component>> TanksClientModel
<<component>> ClientDoer
<<use>> Modifies elements in
<<component>> FightManager
<<component>> Fight ManagerModel
<<use>>
<<component>> FightManagerUI
<<component>> FightManagerController
<<use>>
<<component>> FightManagerDoer
<<component>> Common
<<component>> Controller
<<component>> BackgroundThread
<<component>> Communicator
Has a
<<component>> Doer
<<component>> GunpowderManager
<<component>> GunpowderManagerDoer
<<component>> GunpowderManagerUI
<<component>> GunpowderManagerController
<<use>>
<<use>>
<<component>> GunpowderManagerModel
<<component>> ShellManager
<<component>> ShellManagerDoer
<<use>>
<<component>> ShellManagerModel
<<component>> ShellManagerController
<<use>>
<<component>> ShellManagerUI

The above diagram depicts the components in the system. There are four distinct components in the system:

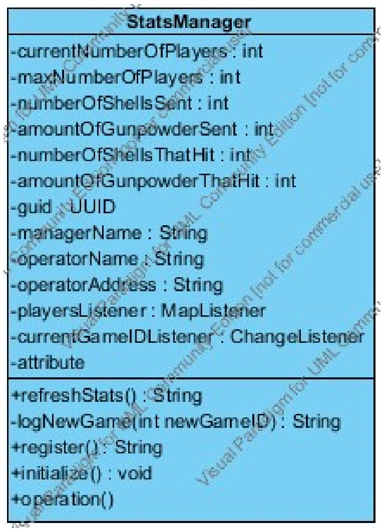
- Client: Responsible for responding to player interaction and responding to requests from managers as to the state of the player.
- Fight Manager: Responsible for tracking players and responding to player requests. Keeps track of certain player stats, and acts as a mediator between players.
- Shell Manager: Responsible for providing shells to players. These shells are empty, and are not available to be fired until filled.
- Gunpowder Manager: Responsible for filling shells provided to it. These shells are filled to a specified percentage, and are returned to the player, ready to be fired.

While the above diagram gives a good idea of which components interact with each, it would be best to describe a general sequence of events that happen. When a communicator sees it has received something on its socket, it tries to form a meaningful message from the bytes. Once a message has been reconstituted, it is placed inside of an Envelope(not pictured) that is stamped with the sender and receiver of the message, and put in the communicator's inputqueue. A Doer continually checks its communicator's inputqueue, checking to see if any envelopes are available to process. When an envelope is available, the message is taken out and the type of the message is determined (i.e. registerrequest, getshellrequest). The Doer then calls the appropriate function to handle the type of message. In all cases, data in the model may be updated, or data in the model queried and that data sent back. If the data is to be sent back, the data is put into an appropriate message, which is put into a stamped envelope that is put in the communicator's outputqueue. In the case of the client, any update made to the model is usually reflected in the UI, as UI components are bound to the elements of the model.

5.4 Statistics Management

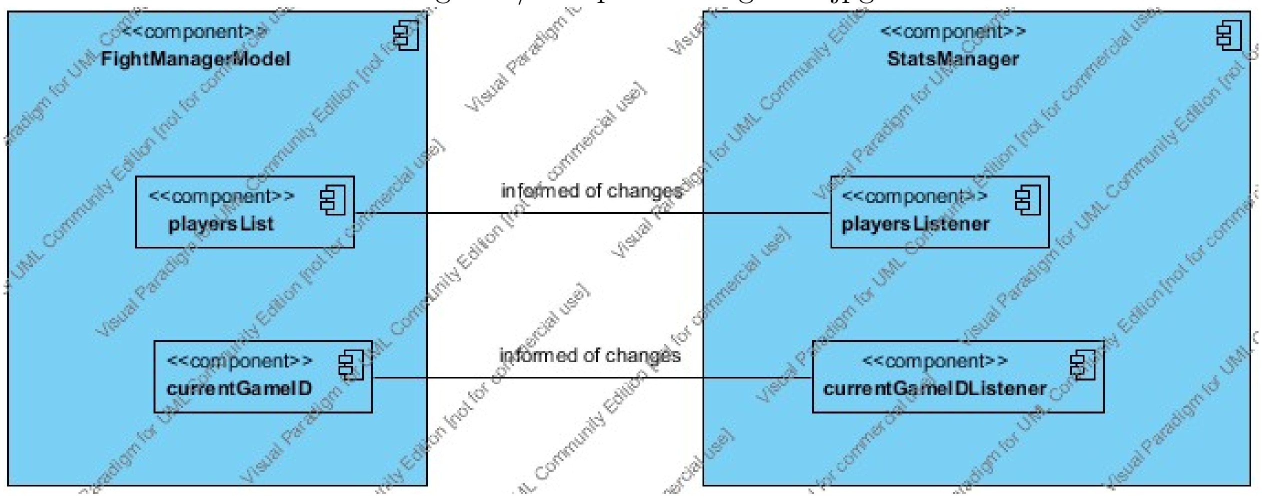
As explained in requirement 12, a fight manager must be able to register with and periodically update a webservice with statistics about the current game. The component(s) used in this process are described below.

Diagrams/StatsManager.jpg


StatsManager
-currentNumberOfPlayers : int
-maxNumberOfPlayers : int
-numberOfShellsSent : int
-amountOfGunpowderSent : int
-numberOfShellsThatHit : int
-amountOfGunpowderThatHit : int
-guid : UUID
-managerName : String
-operatorName : String
-operatorAddress : String
-playersListener : MapListener
-currentGameIDListener : ChangeListener
-attribute
+refreshStats() : String
-logNewGame(int newGameID) : String
+register() : String
+initialize() : void
+operation()

The attributes and methods of the StatsManager, which is the component that aggregates relevant statistics about the game and may be called to refresh those statistics.

Diagrams/ComponentDiagram2.jpg



Described above is the general interaction between the FightManagerModel and the StatsManager. It is with this interaction that the StatsManager aggregates and updates its statistics. The attributes playersListener and currentGameIDListener are registered when the StatsManager is initialized to listen to changes in the FightManagerModel. The needed statistics can be determined as the model is changed by the FightManagerDoer, allowing the StatsManager to be a (mostly) hidden entity. The FightManagerDoer must routinely call refreshStats on the StatsManager, but it does so at its own discretion.

6 Protocol Conversations

6.1 Uniqueness

One concern when communicating these protocols across multiple entities is uniquely identifying conversations within an entity. The strategy employed by this system is to have each entity be assigned a unique process ID. All managers are assigned a process ID when they start up, and all players are assigned a unique ID when they register. An entity wishing to start a new protocol conversation selects a number which uniquely identifies that conversation to them. This unique conversation number and the initiating entities' process ID are communicated with every message in the conversation. Any entity can determine which conversation a message corresponds to by looking at the initiating entity and the conversation number unique to that entity.

6.2 End Points in a Conversation

Another concern is knowing where entities in the system are. This concern is mitigated by a series of progressive steps. As previously described, a client is first known to the system when they register with the fight manager. It is expected in the system that each client knows at startup, without any outside communication, where each manager is located. This allows clients to initiate conversations with every manager. Managers, in order to communicate with clients, track the initiators endpoint in simple request-reply protocols to communicate back to the client. In order to facilitate the communications with clients that did not initiate the conversation, which must occur in the more complex protocols, the fight manager keeps a list of the most recent endpoint used by a client. Whenever the fight manager needs to communicate in this manner, such as when the fight manager must get the last locations of a client, the fight manager uses this list.

6.3 Message Tracking

For each type of conversation there are specific messages that are expected and that contain information vital to the conversation. An object oriented approach is taken in order to organize messages related to a conversation to some type. An object of a base type Conversation is created and kept for each unique conversation number, process ID combination at every endpoint participating in the conversation. Every protocol described previously within the document has specializations of this base type which allows for the types of messages expected within that conversation to be kept within. In this way each conversation has access to all messages relevant to it.

6.4 Message Resends

The specialized conversation types also maintain a timer which tracks how long it has been since the last message was sent without a response. A specific time has been determined for every conversation type of when a response is expected. When a timer expires, the original sent message is sent again. These duplicate messages maintain the same identifying message number, conversation number, and process ID that the original did. As requests in a conversation may be resent, it is also possible that the receiving endpoint receive duplicate messages. In the case that this message corresponds to an active and available conversation, the conversation simply determines whether to reprocess the message or if a reply to the message needs to be resent. In the case a reply needs to be resent, the same reply message previously sent is resent, without modification to message, conversation, or process number. If a message is received that does not correspond to an active and available conversation, it is determined whether it is old. A message is determined to be old when its conversation number is less than our lowest conversation number in reference to the initiating process ID. If it is old, the message is simply ignored and assumed to have taken much longer than expected to get to this endpoint. If the message is not old, a new conversation is created based on what type of message it is, and the appropriate action taken by that conversation.

6.5 Conversation Completeness

Each specialized Conversation type knows the last message expected in a conversation. When a conversation's initiator receives the last message in a protocol and adds it to the conversation, the conversation may then clean itself up by freeing up its memory. Other participants in the conversation are expected to wait a determined amount of time, usually 1 minute, before deletion to ensure the last message they sent was received correctly and that no resends are needed.