Asteroid Miner Game

INF 221 Software Architecture - Winter 2013

February, 13th 2013

[redacted]

Contents

Tables

Figures

1 Game Scenario

"Year 2100, planet Earth is depleted of essential minerals, most dramatically Indio (used in touchscreens), Gold (for digital circuitry), and Copper (for our electric grid)."

1.1 Introduction

With the above motivation in mind we have designed a software architecture for a multi-player, networked Asteroid Lander game with an extra element of interaction through mineral mining and fuel management. This document describes our game design and the architectural decisions made.

1.2 Multi-player Goal

The goal is to explore one Asteroid rich in minerals which are essential to the current resource depleted Earth. Players will cooperate to complete a goal of a predefined amount of minerals. Players will have to land in specific plots of the Asteroid, load the minerals and bring them back to the Base Station. For doing that, the players will share a total amount of fuel available fuel in the Base Station. As the game progresses, players can buy extra fuel by with the minerals they have already collected.

1.3 Glossary

Table 1 Glossary

Term	Description
Asteroid	It has a name, a gravity measure, and a **Mining Grid** (see definition below)
Base Station	It keeps track of the following data: - Fuel shared by all game players - Minerals already collected - Mineral amount targeted (Goal Score) - Grid of exploitable fields in the asteroid - Asteroid being mined

Mining Grid	It keeps track of type and amount of minerals available in each field. Each field has also a limited amount of mineral exploitable per landing.
Spaceship	Provides all the graphic input and output information for the user to play the game. An icon represents the Spaceship and is viewed on a screen. The same screen displays the Asteroid terrain and the gauges for fuel level, velocity, **Leader Board**, field plots available and the **Game Score**.
Leader Board	It keeps track of the spaceships and their respective amount of points.
Game Score	Amount of minerals collected and the **Goal** for each mineral type. The Goal Score is kept updated for all players.
Goal	Total amount for each mineral that players should collect to win the game.

1.4 Real-Time Data Sharing

The multi-player aspect implies sharing resources and information. Resources consist of expendable items (Fuel and Minerals). Information consists of keeping all players aware of important game states (Game Score, Fuel Level, Leader Board).

Players will share two real-time resources:

- <u>The Fuel Reserve in the Base Station.</u> Fuel is spent during landing is consumed from the spaceship during landing. After spending all fuel, Spaceships have to request a refuel from the Base Station
- <u>The Asteroid Mining Field</u>, which is divided in plots by types of minerals

Players will be kept aware of the information updates regarding advancements towards the game goal and also how well their peers are landing. Regarding the latter, after each successful landing, the player earns points as recognition of her skillful piloting. Points are displayed in a **Leader Board** visible in real-time by all players.

1.5 Simple User Story

1. Players join the game;
2. When there are at least 2 players, the game is started;
3. Player selects an available Mine Plot on the Asteroid;
4. After this choice the Spaceship starts descending and consuming fuel;
5. If landing is unsuccessful (i.e. a crash), the player is out of the game;
6. Otherwise, landing is successful, the player earns points and have the Spaceship loaded with the minerals available in the Plot
7. Player can then choose among three options:
   - a Buy fuel to the Base Station
   - b Return to Base Station to Unload the Minerals (
     - i This will also refuel the Spaceship
   - c Quit the game
   - d Select another Mine Plot to land
8. The game terminates when
   - a The goal is accomplished
   - b Either all Spaceships have quitted or crashed

1.6 Graphic User Interface (GUI)

Table 2 Elements of the GUI

Element	
Mine Plot	Visually represents the surface on which the Spaceship will land.
Spaceship	An iconic representation of the Spaceship which will be guided by the player to land on the Asteroid.
Fuel Gauge	Displays the amount of fuel left in the Spaceship at any moment.
Horizontal Speed Gauge	Displays the direction and amount of horizontal speed.
Vertical Speed Gauge	Displays the amount of vertical speed.
Weight	The amount of tons of minerals loaded in the Spaceship.
Game Score Gauge	Displays the up-to-date amount of minerals collected and the game goal.
Leader Board	Displays the name of the Spaceships playing ranked by their earned points.
Available Mine Plots	Displays how many plots are available per mineral type in the current Asteroid.
Notifications	Text to make players aware of changes ("player A crashed", "player B refueled", etc.).

Figure 1 has a mockup of the game.

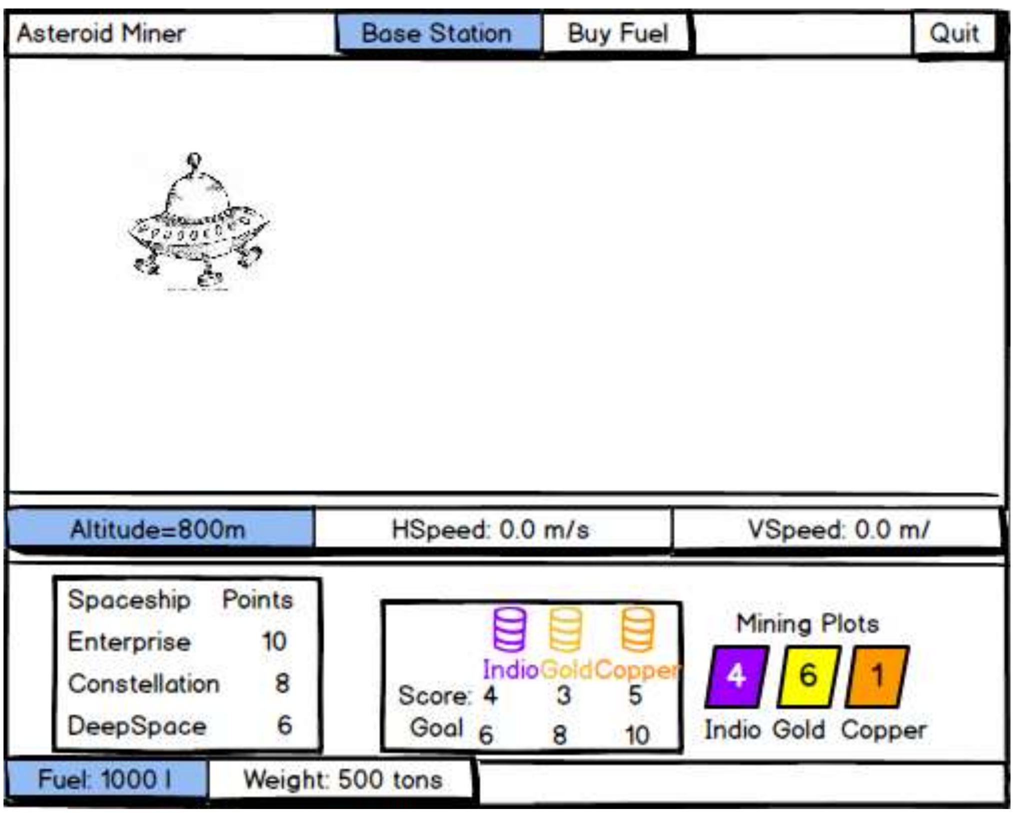


Figure 1 Game Console

1.7 Non Functional Requirements (NFR)

1.7.1 Extensibility for Fun and Engagement

The following extension points were selected as possible extensions to make the game more engaging. It is a real challenge to discovering the right features which would make the game fun and engaging. The future decisions about such extension points depend on detailed usability studies with final users. Therefore, our concern was to enable some extensibility in aspects we find promising. Below are the possible extensions our architecture should be able to support without major disruptions in the overall design.

Asteroid Conditions:

- Change Gravity
- Include Atmospheric Events such As Wind

Spaceship Types:

- Let the user customize her Spaceship by defining attributes of cargo capacity (weight and volume), fuel capacity, speed, and landing system (e.g. parachutes, reverse throttle).

Collective Prizes:

- For each set of Mine Plots exploited the Base Station receives extra fuel. A set of Mine Plots correspond to a specific mineral, therefore, an Asteroid has as many Mine Field SETs as the different types of minerals.

New Types of Minerals:

- Enable the addition of new types of minerals with different densities.

The solution for extensibility is to specify interfaces to enable loosely couple integration between the components of our architecture.

1.7.2 Scalability

The game must support multi-users by replicating clients (Spaceships). The most process intensive requirement expected is the calculation of the Spaceship position during landing. The solution envisaged for that is to replicate in each client the engine responsible for such calculations. Hence, we will have neither network traffic nor server-side concurrency due to the calculation process.

2 Software Architecture

2.1 Choice for Architectural Styles

Table 3 Options considered and respective rationale

Option	Analysis	Evaluation
Peer-to-Peer	Components: GameServer, BaseStation, Asteroid, and Spaceship.	Positive: Handles each entity as an independent actor forcing the establishment of explicit interfaces. Opens the opportunity to have more components distributed in different machines. Negative: Increases the network traffic. Increases the points of failure in the system, since local components would now be accessible via network connections. Increases complexity without an explicit and defensible need for it.
Event-Based	Components: GameServer and Spaceship EventBus: Connector that registers the componets based on the events they are interested to receive.	Positive: Decouples the server and client side of the game while still keep the integration and interaction consistent. Negative: Requires a more complex connector than a client-sever solution.
Blackboard	Shared memory = Mine Fields, Mother Tank Components = Spacecraft and GameServer	Positive: One simple solution to share data. Negative: Has no solution for the shared logic, i.e., keeping track of the number of the active Spaceships in the game and synchronizing the game session.
C2	Level 0 (deepest): Physical engine, Asteroid Mine Fields, Monther Tank Level 1: Spacecraft gauges (speed, fuel, weight) Level 2: Spacecraft speed controls (up, left, right) Level 3: Game ranking	Positive: clear separation of responsibilities among components as well as their interactions. Negative: Replication of the Physical Engine would be more difficult, since it would be a component reused by the upper layers.
Client Server	Client: Spaceship with all the intensive calculations and processing. Server: GameServer with the shared date	Positive: All intensive processing and state update could be implemented in the client. Meanwhile the shared data with less frequent update cycles could be maintained at the server side. This would make the game simple, scalable and elegant to implement. Negative: Any change in the clients' needs

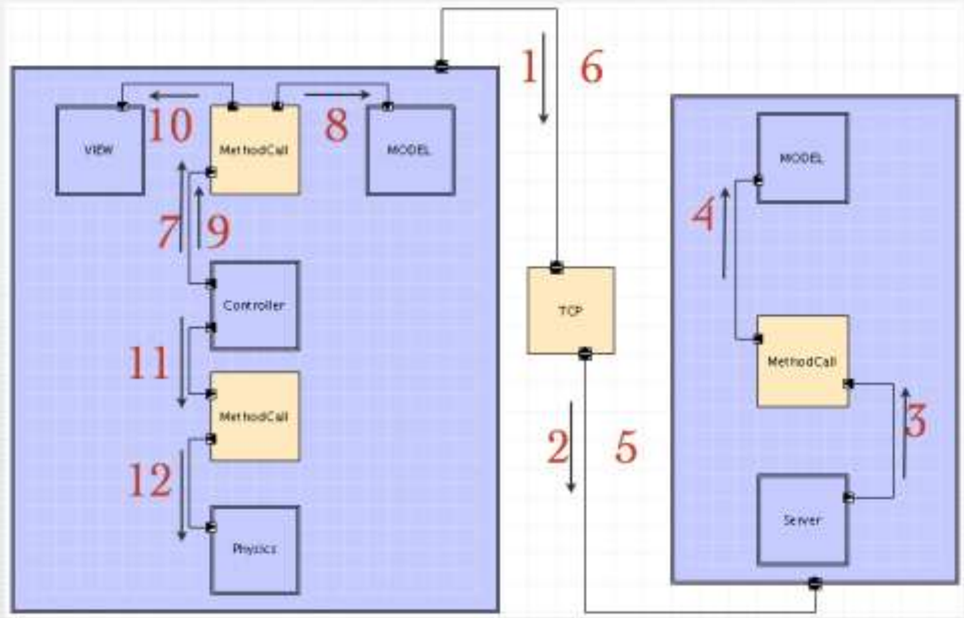
		(e.g. a new share status) would impact the existing client and server implementations. I.e., client and server are more tightly coupled than in an event-based solution.
Publish-Subscribe	Components: GameServer and Spaceships	Positive: Extensible for new events and Spaceships with different needs in terms of sharing data with the server and the other Spaceships (not our current case) Negative: Increase the complexity of the connector and the communication overhead between Spaceship and GameServer.

2.1.1 Final Design Choice

Client-Server Architecture with a connector that uses events to properly discriminate the various types of interactions. Such connector will mediate all the communication between each Spaceship and the GameServer. There will be a connector instance for each one-way interaction between a Spaceship and the GameServer (and vice-versa).

Spaceships and the Server do not register to listen to events, because all events needed by the both sides compose a fix set and are known in design time. One extension point suggested is to have different Spaceships, which in turn would comprise complementary sets of events. In such situation, an extension to the connector would be required.

Figure 2 depicts the high level components and connectors of our architecture as well as an enactment by means of a sequence of interactions. The next section details each the components and connectors showed in Figure 2.



Once the game start Client (Controller) start pushing their game actions to the server through the TCP connector. Server reads this data and maintains the state (in memory) of every player using server side models. Server pushes the game data to the TCP connector and the clients(controller) get this information. Controller updates the client side models with the new information and then updates the view with the new model data. Controller, also receives inputs from user actions and calls Physics component to get data to update spacecraft speed and direction.

Figure 2 Diagram with Control Flow for the Chosen Architecture

2.1.2 Game Platform -

Platform: Astral/Pygame
Programming Language: Python

The choice of programming languages, graphics and physics engines and networking libraries impact many components of the development process. While selecting our tools, we were aware of time constraints (1 month at design-time), robustness of networking features, game engine stability, and availability of community support and documentation.

Pygame (http://pygame.org/wiki/about) is a widely used, stable, and user-friendly game development framework written in Python. The *Pygame* community frequently holds week-long game competitions, which is a testament to the kind learning curve and rich features of *Pygame* framework. Python is a highly expressive language with many useful packages, and is a good choice for rapid development projects like ours.

While researching *PyGame*, we encountered numerous networking libraries to implement multiplayer functionality. *Mastermind* and *PodSixNet* are most commonly used for multiplayer networked games, but are both fairly low level. We also considered a *Tornado* server and *Websockets*, but we acknowledged that such an elaborate solution is more time-consuming and bug-inducing than our limited timeline allows. *Astral Networking* (http://code.google.com/p/astralnetworking/) is a high-level Python networking tool that provides the functionality we need to build our Lander game. The system is built on top of the *Mastermind* and *PodSixNet* libraries (used interchangeably as its adapters), and provides an out-of-the-box implementation of networked multiplayer with a few examples. Nonetheless extensible and used in numerous *PyGame* projects, it has limited documentation. Hence, we plan allocate more networking and synchronization development time to make sure we work out bugs. We would need to build the same functionality if we used *Mastermind* or *PodSixNet* (both quite stable), so Astral is a very handy and *PyGame*-compatible head start. We elaborate on our design decisions as a result of features of this framework later in the document. One of our team members has developed games in *PyGame* previously and can provide insight on limitations and strengths of that platform, as needed.

We seriously considered Java due to the mapping feature of the 1.x tool, and our team's extensive experience with the language. However, there is a limited quantity of robust and stable Java game development frameworks, and it is a difficult language in which to develop UI features. By using *PyGame* and the *Astral* engine (developed by a PyGame developer specifically for the framework), we will be able to create a functioning multiplayer game interface within a week of our initial architectural style design meeting. In the event that we encounter networking issues with the *Astral* tool, the Python *Tornado* webserver is a flexible, stable, and well-documented alternative choice.

Ultimately, by balancing our combined experience levels and our cautious, we have selected a framework system that provides rapid development functionality to bring our architectural design to life.

2.2 Architecture Description

2.2.1 Components and Connectors

The client side is implemented as Model-View-Controller pattern. Each of the components and connectors are described below. Just below the figure, there is a legend with the names for each element. Each name is **traceable** to the Class Diagram in available in Figure 3.

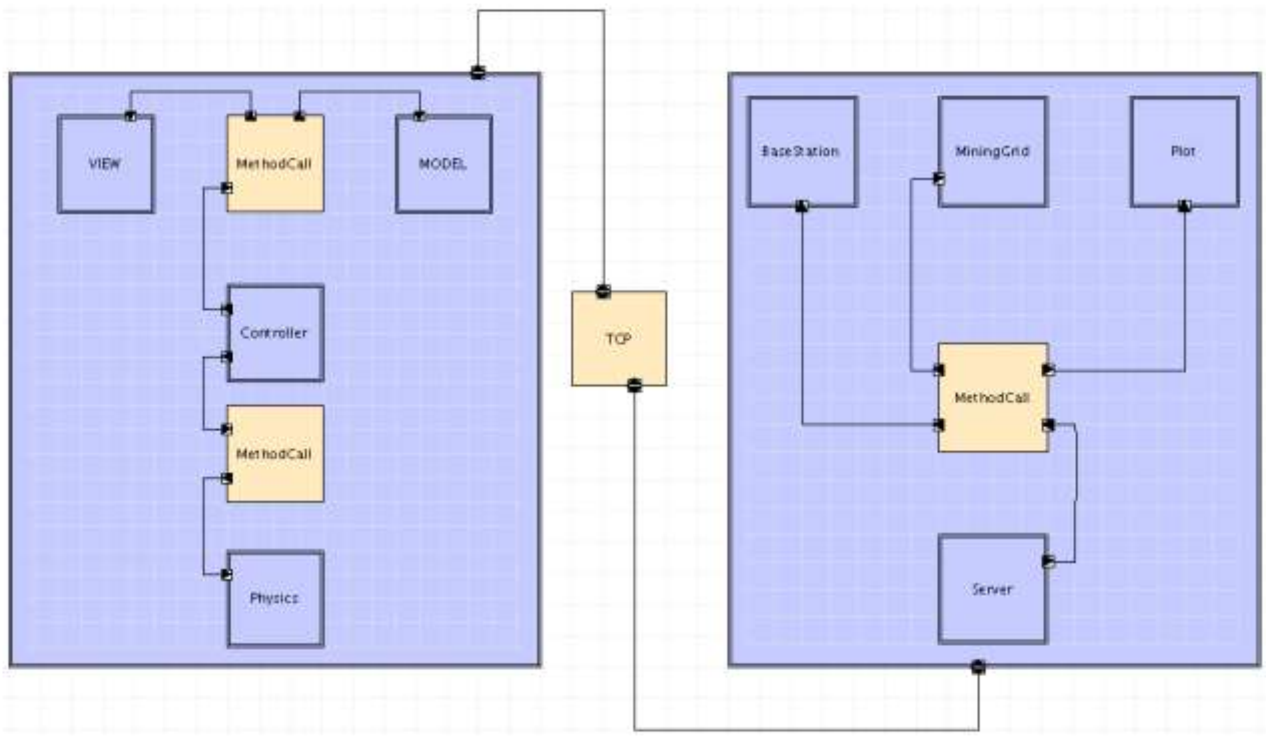


Figure 3 Architecture Diagram created in ArchStudio-4

Table 4 Legends for Figure 3

Component Connector (xADL)	Description	Class Diagram (UML) Figure-3
MethodCall	Integrates via method calls	MethodCallConnector
TCP	Integrates via TCP calls	TCPConnector
Controller	Handles events generated by the GUI and updates the GUI with data retrieved from the GameServer	SpaceshipController
View	Manages the GUI	SpaceshipViewer
Server	GameServer keeps track of data shared among players	GameServer
Physics	Performs the calculations needed to correctly process the motion of the Spaceship in the screen.	PhysicsEngine

2.2.1.1 Detailed Description of Connectors

TCP connector

Role: Communication

Type: Procedure Call Connector, Facilitator

Client and server both have an instance of this connector and they call the `send()` method of this connector to push data to the other side.

Parameters: Host, Port, Data, Communication protocol. (default = 'TCP'), Network Adapter (default = 'podsixnet')

Podsixnet library is a lightweight network layer that asynchronously serializes network events and arbitrary data structures, which are delivered them via the TCP connector. The TCP Connector in turn delivers the data to the required components. The TCP connector will be an instance of podsixnet and will act as a facilitator.

MethodCall Connectors

Role: Communication

Type: Data Access Connectors

These are procedure call connectors. Different components have their instances and these components call the required methods of these connectors to access data from other components.

2.2.2 Evaluating the Design Solution

2.2.2.1 Extensibility

We provided extensibility of the physical engine and the shared data server implementations. It was accomplished by a layer of services which decouples the client from the specific implementations.

2.2.2.2 Network Usage

Our architecture uses the network resources with parsimony due to the following design decisions:

- Spaceship physical computations happen in client side
- Shared data is only transmitted in an event of change on it.
- No need for a unified clock, because players are synchronized by means "state change events"

2.2.2.3 Game synchronization

Since it is a multiplayer game, we have the following two types of synchronizations among players:

- Shared data is kept up to date for all players via "state change events"
- Players are made synchronously aware of the events of game start and finish

2.2.3 UML Class Diagram

The UML Class Diagram (Figure 4) provides a second level of detail to the System Architecture diagram we saw in Figure 3. In the UML Class Diagram we can see the methods and attributes for each component and connector. Moreover, some auxiliary data structures are also made explicit, such as the Position, the BaseStation, the MiningGrid and the Plot.

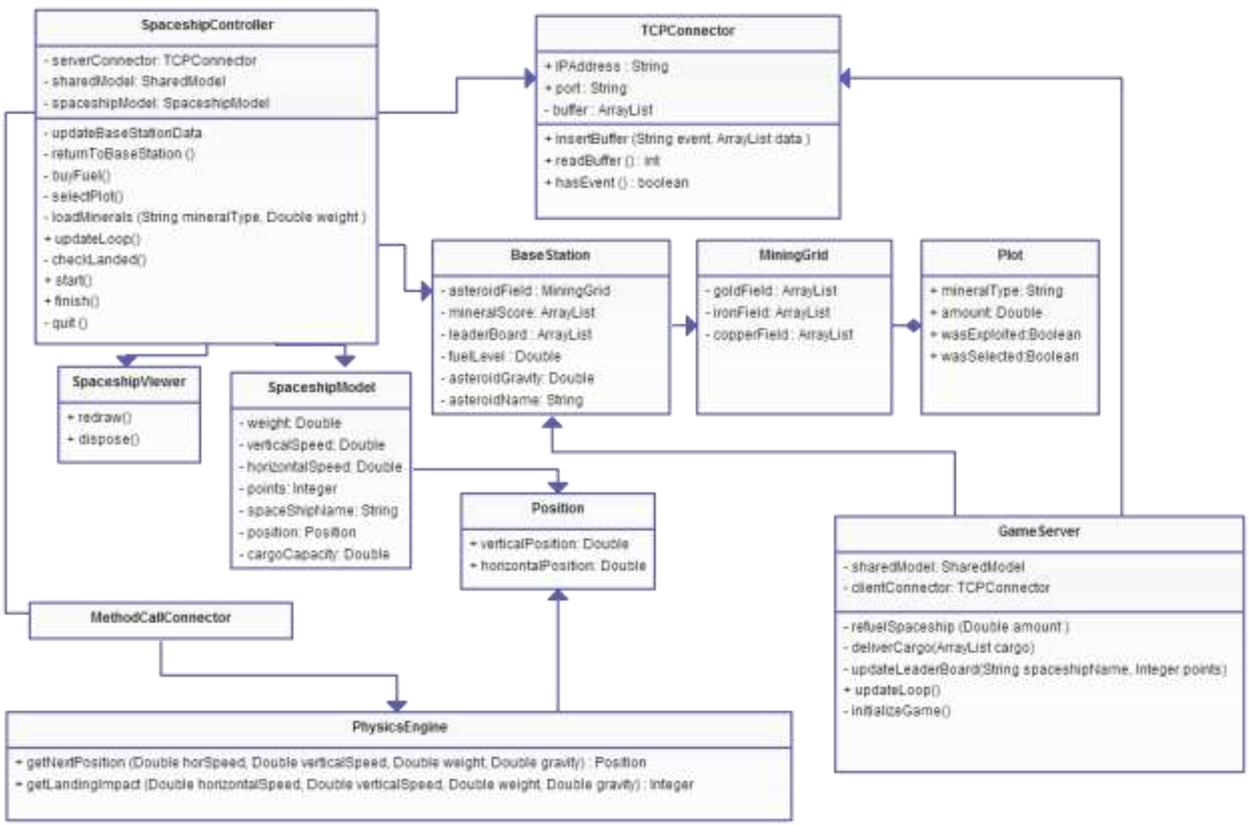


Figure 4 UML Class Diagram

2.2.4 UMLState Diagram

Figure 5 depicts the state machine of the Spaceship. This simplicity of this state machine contrasts with the amount and complexity of events exchanged between the client and the server (shown in Table 5).

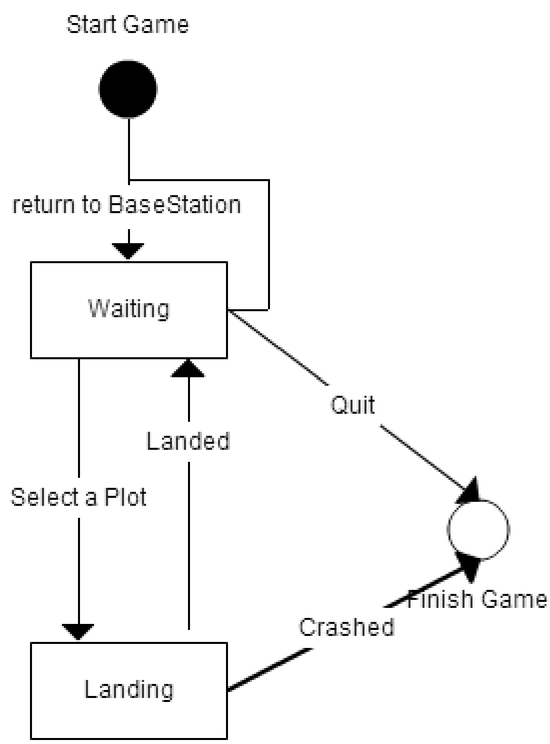


Figure 5 UML State Diagram with the Spaceship states

2.2.5 Activities and Events Exchanged

Complementary to the Spaceship state machine, we also designed the events shared between the client and the server. For that we used a Graphical AADL Notation (Figure 6 and Figure 7) depict the sequence of such events in the context of the loops in the client side and in server side.

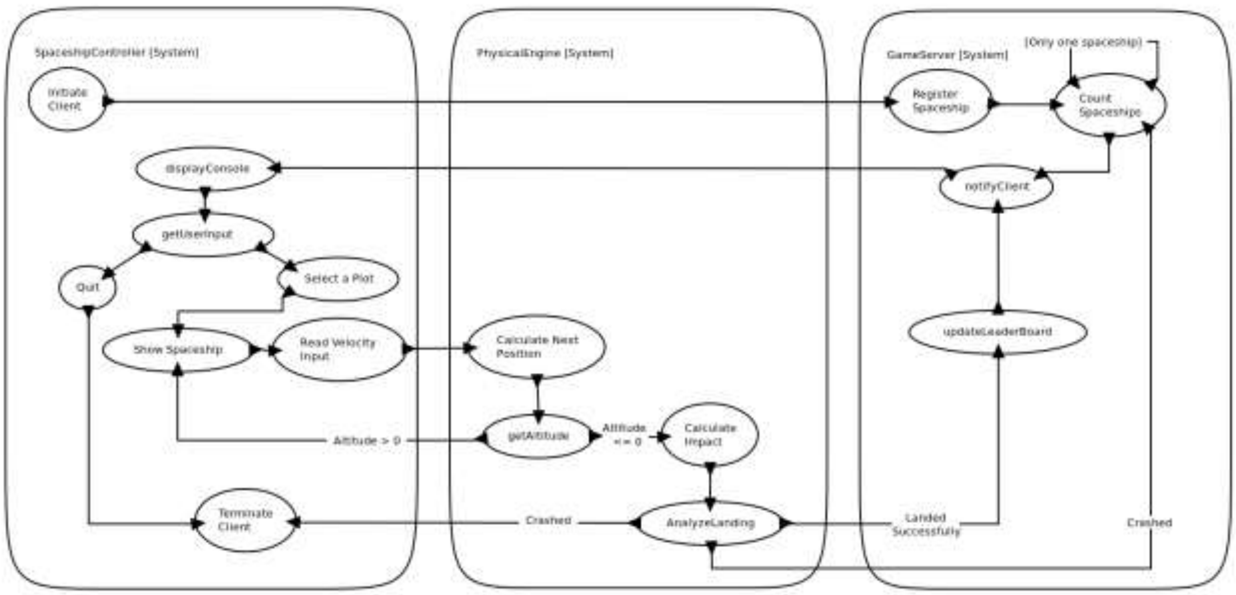


Figure 6 Landing Loop

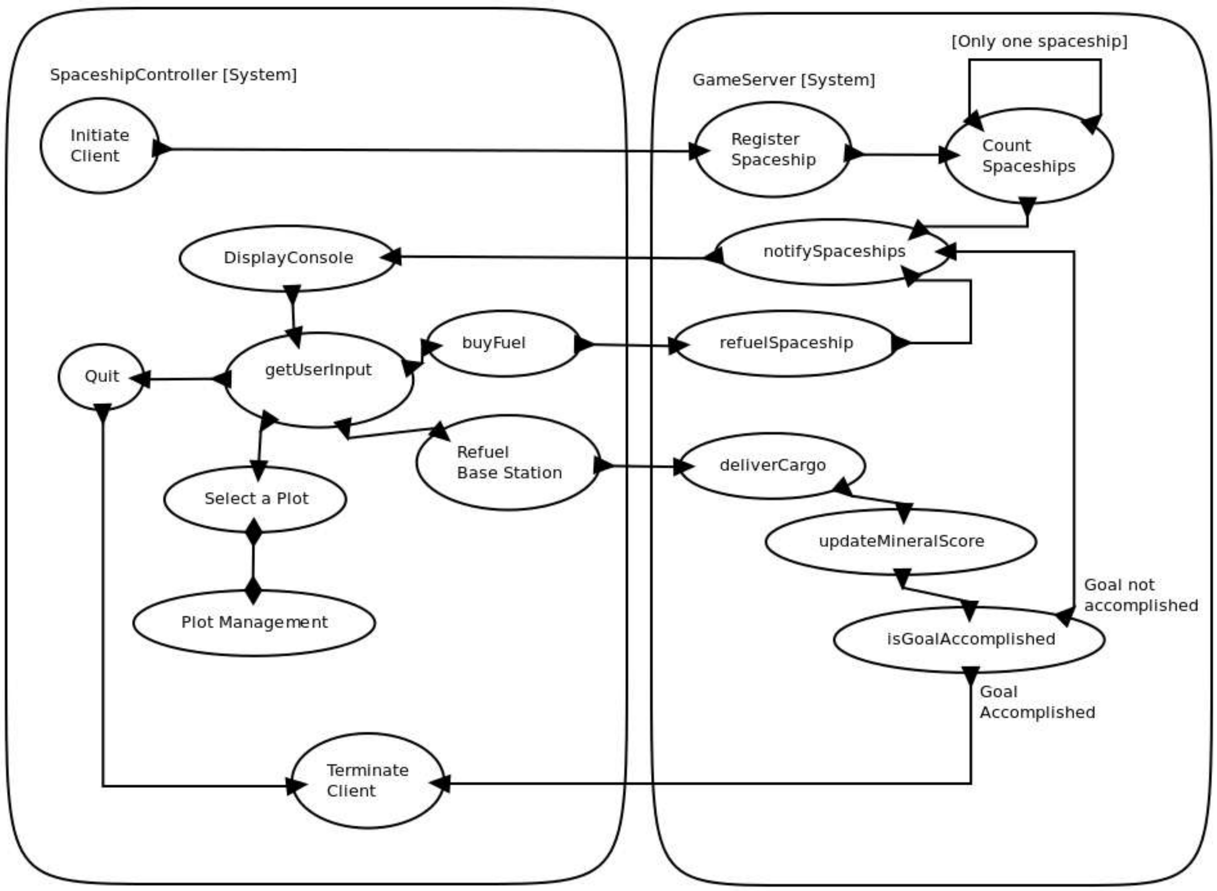


Figure 7 Server Loop

2.2.6 UMLSequence Diagrams

In order to demonstrate how the methods in the class diagram generate the sequence of events describe in the AADL diagram, we created two UML Sequence Diagrams (Figure 8 and Figure 9).

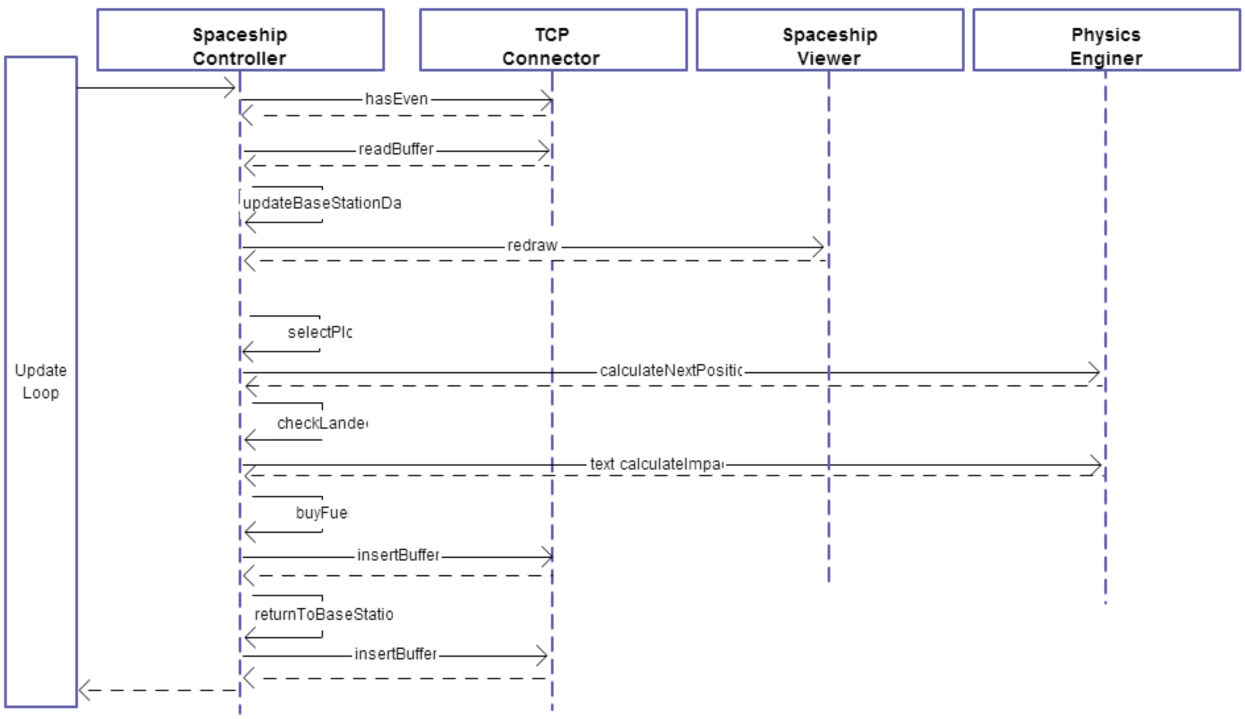


Figure 8 Sequence of method calls in the client side

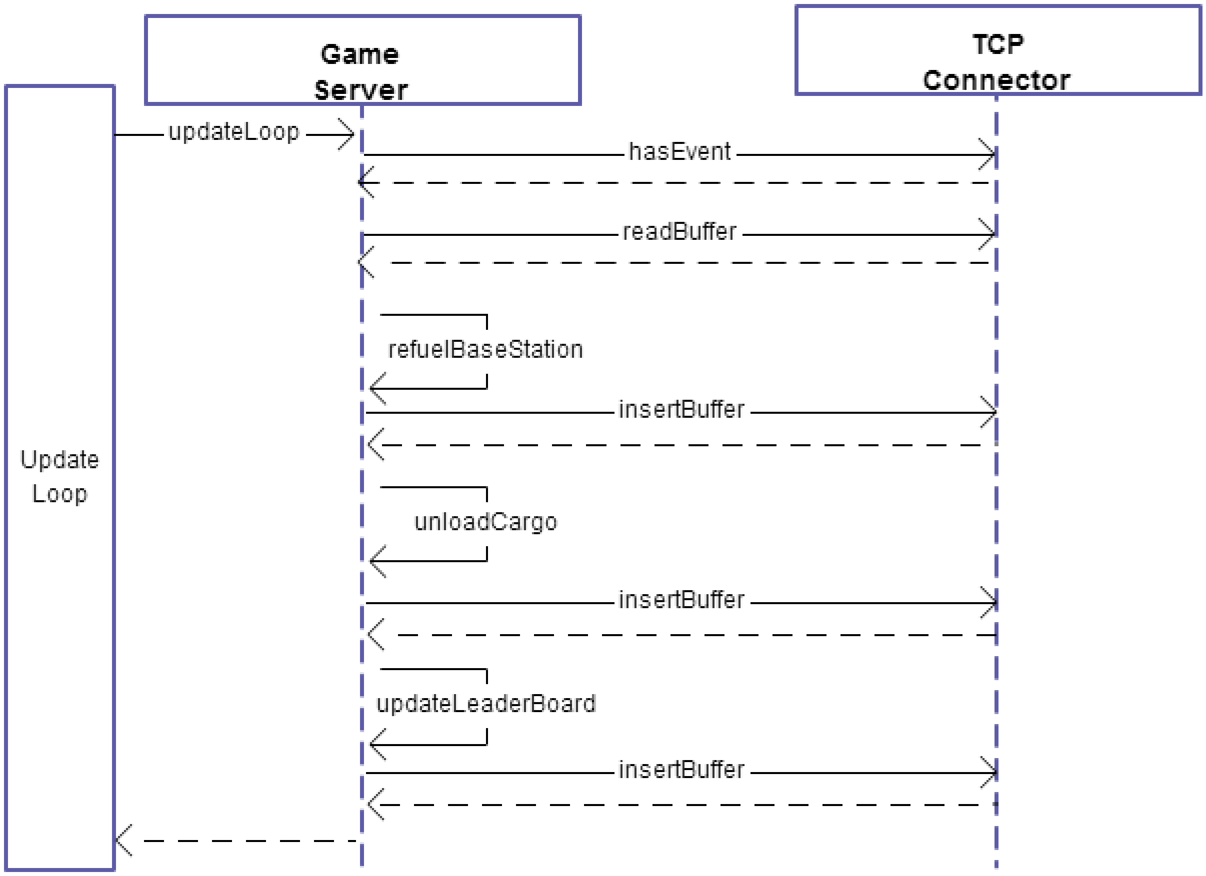


Figure 9 Sequence of method calls in the server side

2.2.7 Detailed Description of State Changes

Some of our “state changes” trigger events while others do not. The reason is that some changes impact solely the client that generated the event. For example, the event of Spaceship landed impacts only the respective Spaceship client. In our game logic this information is not necessary for the other players. On the other hand, when a Spaceship requests to buy fuel for the BaseStation, this event affects the level of fuel seen by all the other players, hence a state change event must be raised. The following table describes all events we will have in our game.

Table 5 Game Events

Name of Event	Source	Trigger	Listener and Methods
Land Spaceship	Spaceship Viewer	LandButton = = clicked	SpaceshipController
Spaceship landed	Spaceship Viewer	(Altitude == 0.0) AND (VerticalMomentum <= 50.0) AND (HorizontalMomentum <= 20.0)	SpaceshipController Method: LandedSafely()
Spaceship crashed	Spaceship Viewer	(Altitude == 0.0) AND (VerticalMomentum > 50.0) OR (HorizontalMomentum > 20.0)	Spaceship Controller, LeaderBoard Method: CrashLanded()
Return to Earth	Spaceship Viewer	EarthButton = = clicked	Spaceship Controller, BaseStation Method:ReturnToEarth()
Buy fuel	Spaceship Viewer	BuyFuelButton = = clicked	GameServer Method: BuyFuel()
Choose plot to land	Spaceship Viewer	(PlotType = = available) AND (Plot Type clicked)	GameServer Method: RequestPlot
Quit game	Spaceship Viewer	QuitButton = = clicked	GameServer Method:QuitGame()
Game Start	GameServer	(NumberOfPlayers > = 2) AND (StartFlag <> 1)	SpaceshipController Method: Network_response()
Game Over	GameServer	(NumberOfPlayers < 2) AND (StartFlag == 1)	SpaceshipController Method: Network_response()
Base fuel changed	GameServer	FuelLevel.add(value) OR FuelLevel.subtract(value)	SpaceshipController Method: Network_response()
Leader Board changed	GameServer	leaderList.update(Spaceship, point)	SpaceshipController Method: Network_response()
MiningGrid changed	GameServer	grid.setFieldAvailable (latitude, longitude, 0)	SpaceshipController Method: Network_response()

3 Part-2 Comments on the Methods and Tools

3.1 xADL Modeling Methods and Tools

Learning about Modeling: One of the most important lessons we learned is that modeling demands the mastering of the tools. The designer will always lack the confidence unless she doesn't know how the various components or connectors can be manipulated with the tool.

Besides that, the modeling experience showed us how options are generated and discarded in the process. Awareness and knowledge of the platform improved as we advanced. We initially designed the architecture as a hybrid of blackboard and events based system. Soon, when we started studying about *Pygame* and *Astral* networking library, we realized that our decisions were too different from the standard model offered by the game platform.

Modeling Notations: Archstudio has options to create structures representing parts of a system. We are using three structures, one for client, one for server and one to integrate client and server. We came across the following elements while designing our architecture:

- <u>component</u>: can be used to represent any component in the system
- <u>connector</u>: can be used to represent any connector in the system
- <u>interfaces</u> : define how a connector or component will interact to the outer system. It has four different options, in/out/none/inout.
- <u>links</u>: links are to join of two interfaces.
- <u>substructures</u>: these are very important if you want to depict hierarchy in the system.
- <u>types</u>
    - <u>component type</u>: to make use of the substructure, we need to create a component type and then assign a structure to the type. We created two component types - client and server.
    - <u>constructor type</u>: similarly we can make custom constructor elements using the types.

Tool - ArchStudio 4:

Installation: We enjoyed the Archstudio 4 for most of the time during this exercise. We faced some difficulties while installing the plugin of Archstudio 4 on Eclipse Juno. The error messages pointed out that there is a problem with the version of Eclipse and Archstudio, as they were not compatible. After updating the Eclipse we could easily install Archstudio on our machine.

Learning Curve: Nearly everything on Archstudio is done through context menu. We took time to understand, and we think we are still not very comfortable with this approach. The grid background is something that cannot be enabled from the context menu. So, we had to go through the series of button clicks exploring our way to enable grid view. The tutorial available online is very limited and also we think there is not enough material (tutorials) available to understand different editors and their uses in Archstudio.

Bugs/Issues: we still have not yet figured out how to draw a link that is parallel to the horizontal lines or to the vertical lines in the grid.

Creating Substructures: while getting familiarized with Archstudio, we created some components and connectors and also fiddled around with different editors. But soon we realized that doing modeling with one level of structures was very straight forward. The majority of our time was spent in learning how to create hierarchy of structures. We are still trying to understand if it is possible to create a link that joins a sub-structure component to a connector that is outside the parent structure. We really wish we could open the xml for structures and directly make modifications to the xml.

Font size: we are not sure if it is possible to modify the font size of the description of components and connectors as we have not yet figured it out.

Future work: we need to learn about different editors that come with Archstudio. So far we only tried Archipelago and ArchEdit.

xADL 2.0: Archstudio generates xADL behind the scenes and it designers are required to know xADL to work with Archstudio.

3.2 UML Modeling

Tool: Creately (www.creately.com), which is a cloud-based tool running inside a browser. The tool enables multiple users to edit one diagram at same time. We have been using this tool in the last two years without any issues. The tool also has a catalog of sample models that are very illustrative for beginners.

Learning about Modeling: UML has a minimalistic approach to diagrams, which makes the learning curve very smooth. On the other hand, such simplicity leaves excessive responsibility to the user to guarantee consistency and coherence among the depicted models. Based on the industrial experience of one our team members, we could realize the pitfalls our team faced while modeling with UML.

Modeling Notation

Concerning coherence, we had issues with the labels in the diagrams, because such properties are not enforced by the language. For instance, in the activity diagrams we must use sentences denoting actions instead of substantives. In the state diagrams we must use substantives to denote states and actions in the arrows to denote change.

Concerning consistency, we had to manually verify whether method names in the classes matched the activities and method calls in the sequence diagram. Further complicating the job, any later change implied in a huge impact on reviewing all the diagrams.

In our opinion UML seems useful to rapidly sketch ideas and to plan ahead the implementation. We would neither use UML to validate/verify our model against our requirements nor to generate code from it. Since the effort to create an UML diagram is low, the cost of discarding it is also negligible.

3.3 AADL Modeling

We chose AADL to model the interactions and control flow between elements of the game loop, game server, and clients. AADL may be installed through a number of toolkits, though their stability and ease of use varies. We used multiple versions to find an optimal one - a combination of TOPCASED with its *Adele* editor proved most streamlined. Documentation on AADL is extensively example-focused, particularly in the embedded systems and mission-critical engineering domains (much more complicated and device oriented than a Lunar Lander game). Some tutorials exist but finding a concrete guide to AADL components is practically impossible.

AADL is designed to be extensible as a modeling language. While it is indeed possible to customize, the core language appears oriented to the software transactions and control flows between hardware components. It was challenging to portray the control flow and particularly decision junctions in our Lunar Lander game with the core elements of the language.

AADL components don't map clearly to gameplay configurations and multi-option game menus; a simple decision tree or multi-branch junction gets messy. However, the simplicity of the language made modeling the overall flow of control between the game server and lander actions smooth. Many AADL editors/modelers use layered systems to organize the graphic editor view. We found AADL easier to program than to diagram in its core language form once we understood the syntax. Many AADL toolkits are dependency-heavy *Eclipse* plugins; a single one (*Ocarina*) that we found had adequate Vim/Emacs integration. A good workflow for future modeling activities would be a *Vim* or *Emacs* plugin with hooks to a simple diagram modeling UI such as *Dia* (which has a user-friendly AADL diagraming feature, actually).

3.4 Preliminary Conclusions

Tool mastering was definitely paramount. By which we mean both the tool and the diagraming language. We realized that learning while doing has its challenges. Since it is difficult to discern between the situations we are learning with the tool or about the tool, we have to accept initial imprecisions and work iteratively as our comprehension and skill together evolve.

Modeling forced us to think about how various components and connectors would interact and also gave us some insight about which properties our system would exhibit. For example, initially we thought of using a database to maintain the game state, but later understanding that the states could be shared at the server side, we confidently dropped the database idea. This also implied in a compromise. On one hand, we would have lesser complexity by stripping out a persistent data access layer and all its database connectors. On the other hand, we would decrease scalability, since the server will now be keeping the complete game state in memory. Hence, our system would be at closer limit to the maximum number of players we could host.

Another similar important decision was sharing game data and not the visual space among the different players. This decision does not have any impact on the architecture of the system but it saved us a lot of accidental complexity related to drawing objects of one player on the screens of all the other players. Moreover, we realized that such operations would require a lot of synchronization triggered by every movement made by one player and the respective and necessary propagation to all other players. Such would end up in slowing down the user experience.

Ultimately, concerning the level of detail provided to the implementation phase. The use of multiple diagrams and the experimentation with three different modeling languages helped us stress aspects we would probably only detect during code integration and testing. Holding us back from diving in implementation issues definitely paid back not only in a more elegant (consistent + coherent) design but also in less uncertainty during the near future implementation.

Asteroid Miner Game - Parts 3,4,5

INF 221 Software Architecture - Winter 2013

[redacted]

"Year 2100, planet Earth is depleted of essential minerals, most dramatically Iron (used in civil engineering), Gold (for digital circuitry), and Copper (for our electric grid)."

[redacted]

Contents

Tables

Figures

1 Introduction

1.1 Objective

This document describes the following:

- The architectural models revised, screenshots and documentation based on the previous report.
- A description of our implementation
- Evidence that the implementation and the model(s) are consistent.

For each section we comment how the implementation reifies the design. We accomplished that by means of tracing each element to the classes and methods.

Just to remember our game logic, below is a synthetic description of it.

1.2 Simple User Story

1 Players join the game;
2 When there are at least 2 players, the game is started;
3 Player selects an available Mine Plot on the Asteroid;
4 After this choice the Spaceship starts descending and consuming fuel;
5 If landing is unsuccessful (i.e. a crash), the player is out of the game;
6 Otherwise, landing is successful, the player earns points and have the Spaceship loaded with the minerals available in the Plot
7 Player can then choose among three options:
  - a Buy fuel to the Base Station
  - b Return to Base Station to Unload the Minerals (
    - i This will also refuel the Spaceship
  - c Quit the game
  - d Select another Mine Plot to land
8 The game terminates when
  - a The goal is accomplished
  - b Either all Spaceships have quitted or crashed

2 Revised Architecture and Mapping to the Implementation

2.1 How our Glossary Mapped to our Entities?

For each of the Glossary entries we provided the class (3rd column in Table 1) or data structure that implements it in the code.

Table 1 Entities and Implementation Classes

Term	Description	Entities in Code
Asteroid	It has a name, a gravity measure, and a **Mining Grid** (see definition below)	The gravity was modeled in LanderSprite class which is in LanderCanvas.py
Base Station	It keeps track of the following data: - Fuel shared by all game players - Minerals already collected - Mineral amount targeted (Goal Score) - Grid of exploitable fields in the asteroid - Asteroid being mined	BaseStationModel.py
Mining Grid	It keeps track of type and amount of minerals available in each field. Each field has also a limited amount of mineral exploitable per landing.	MiningGrid data structure in class BaseStationModel.py
Spaceship	Provides all the graphic input and output information for the user to play the game. An icon represents the Spaceship and is viewed on a screen. The same screen displays the Asteroid terrain and the gauges for fuel level, velocity, **Leader Board**, field plots available and the **Game Score**.	We have two classes: SpaceshipViewer.py and SpaceshipController.py
Leader Board	It keeps track of the spaceships and their respective amount of points.	LeaderBoard data structure in class BaseStationModel.py
Game Score	Amount of minerals collected and the **Goal** for each mineral type. The Goal Score is kept updated for all players.	GameScore data structure in class BaseStationModel.py
Goal	Total amount for each mineral that players should collect to win the game.	Goal was model as constant variables in the Constants.py

As observed in Table 1, some entities mapped to classes, others to data structures and other to simple attributes.

2.2 We missed the Constant Values in our Design!

Something we **did not predict** in design time was the need for initializations and constant variables. Two sets of constants were needed. First the initial values for fuel, goal, conversion rate from gold to fuel, number of plots available for each mineral, etc. The sheer amount of constants emerged only when we got into the details of the methods. Second, we also needed standard labels (strings) to reference the events unambiguously in the client and the server, so each side is confident about what data and action is expected.

The solution was to place all of them in a class **Constants.py**. This class is shared by the server and the client. Below is the list of constants adopted:

2.3 How did we implement the real-time data sharing?

The multi-player aspect implies sharing resources and information. Resources consist of expendable items (Fuel and Minerals). Information consists of keeping all players aware of important game states (Game Score, Fuel Level, Leader Board).
Players will share two real-time resources:

- The Fuel Reserve in the Base Station. Fuel is spent during landing is consumed from the spaceship during landing. After spending all fuel, Spaceships have to request a refuel from the Base Station.
  - Implemented by class BaseStationModel.py, attribute “fuel”
  - Maintained by the following methods in GameServer.py
    - canRefuelSpaceship
    - withdrawFuel
    - canBuyFuel
    - buyFuel

- The Asteroid Mining Field, which is divided in plots by types of minerals
  - Implemented by class BaseStationModel.py, attribute “MiningGrid”
  - Maintained by the following methods in GameServer.py
    - canAssignPlot
    - freePlot
    - assignPlot
    - conquerPlot

Players will be kept aware of the information updates regarding advancements towards the game goal and also how well their peers are landing. Regarding the latter, after each successful landing, the player earns points as recognition of her skillful piloting. Points are displayed in a **Leader Board** visible in real-time by all players.

- Implemented by class BaseStationModel.py, attribute “LeaderBoard
- Maintained by the following methods in GameServer.py
    - getLeaderBoard
    - getPlayerScore

2.4 Graphic User Interface (GUI)

Next in Figure 1 we compare the mockup with the actual interface.

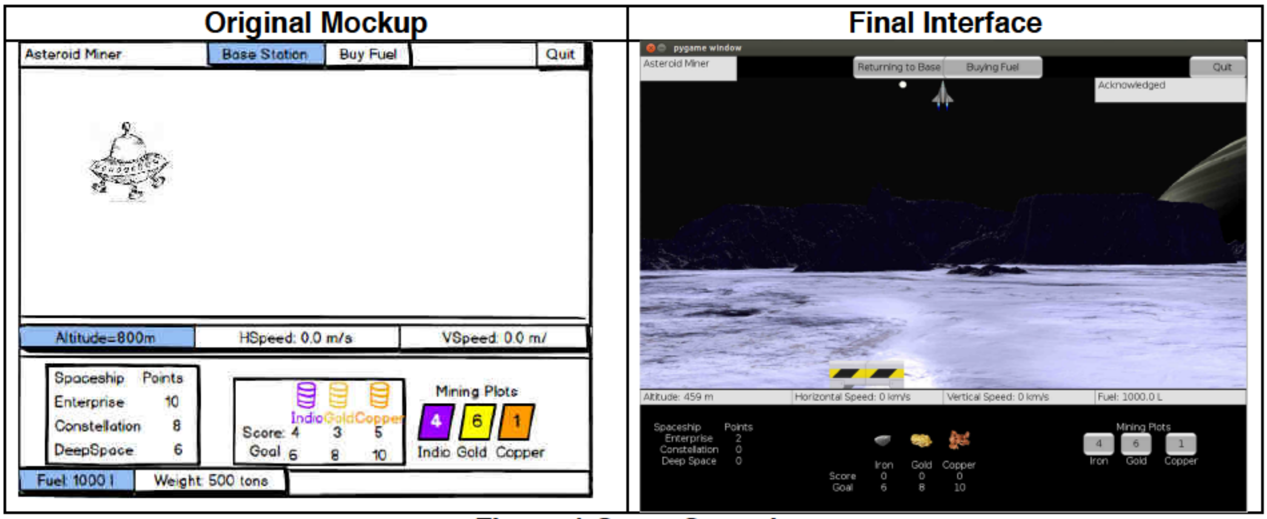


Figure 1 Game Console

Below we describe how each GUI element was coded in the interface and the respective classes in source files. We used three classes to implement the GUI elements:

- LanderContainer.py = to keep track of the event handling
- SpaceshipViewer.py = to keep track of the game loop
- GameDataPanels.py = to keep track of the GUI model

Table 2 Elements of the GUI

Element	Description
Mine Plot	Visually represents the surface on which the Spaceship will land.
Spaceship	An iconic representation of the Spaceship which will be guided by the player to land on the Asteroid.
Fuel Gauge	Displays the amount of fuel left in the Spaceship at any moment.
Horizontal Speed Gauge	Displays the direction and amount of horizontal speed.
Vertical Speed Gauge	Displays the amount of vertical speed.
Weight	The amount of tons of minerals loaded. **(REMOVED)**
Game Score Gauge	Displays the up-to-date amount of minerals collected and the game goal.
Leader Board	Displays the name of the Spaceships playing ranked by their earned points.
Available Mine Plots	Displays how many plots are available per mineral type in the current Asteroid.
Notifications	Text to make players aware of changes (“player A crashed”, “player B refueled”, etc.).

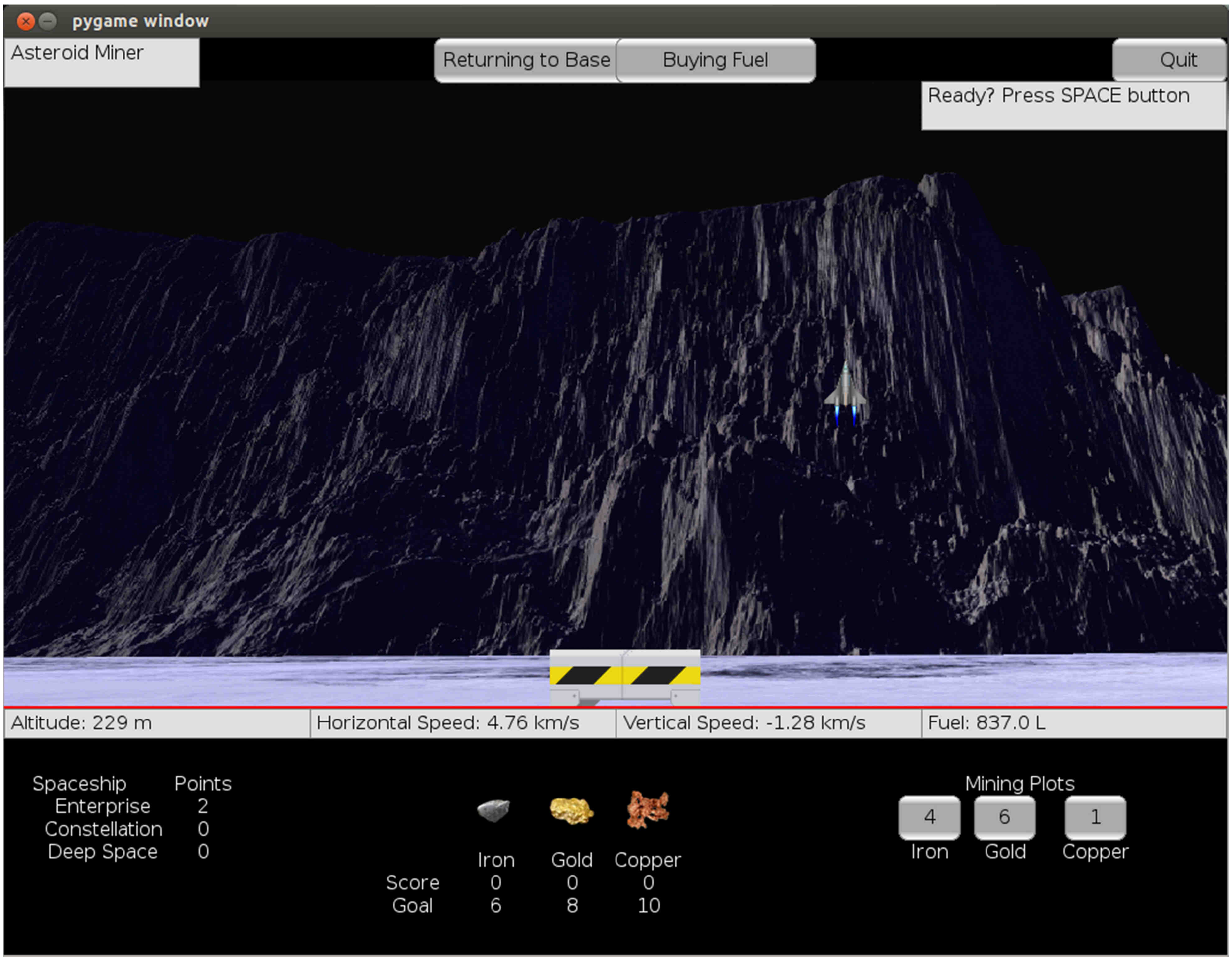


Figure 2 GUI with Spaceship Landing above Crash Speed (red line on the bottom)

2.4.1 So, what is New in the GUI?

We had two changes made in the UI. First, the inclusion of a notification text area – see on top right corner of

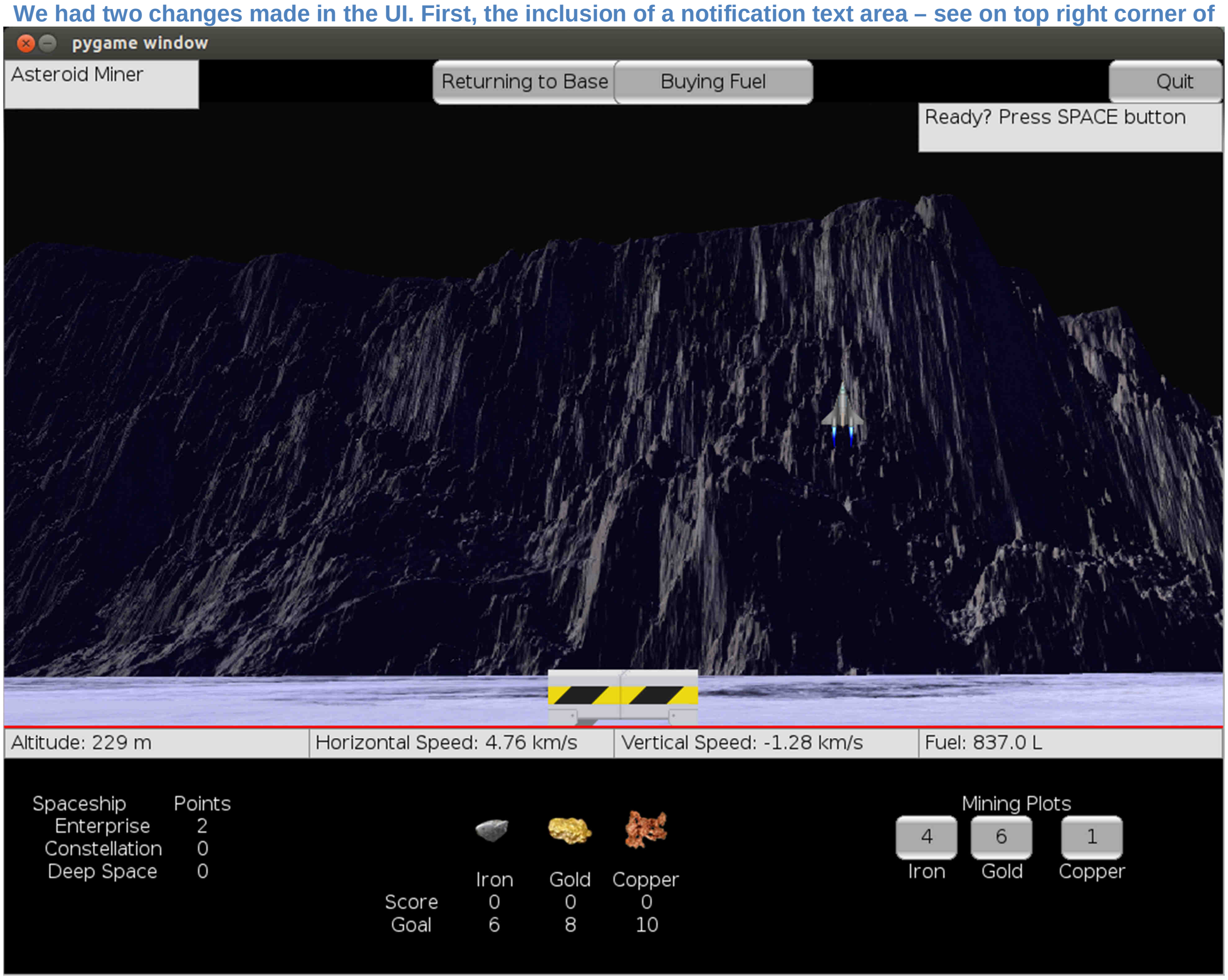


Figure 2. During functional testing we realized that, since we have a multi-player game and the UI does not provide a visual representation of the other the state of the other players, it was very difficult to coordinate with other players (towards the collective goal). The solution we found was quite simple, reuse the event messages which were already part of our architecture and display them. Examples of relevant messages for the user are:

- Base Station is running out of fuel
- Some player just bought fuel to the Base Station
- Some player quit or crashed
- Goal was accomplished

The second change was to remove the Weight because the PhysicalEngine ended up not needing it to calculate the Spaceship position.

2.5 How did we (and did not) implement the Non Functional Requirements?

2.5.1 Extensibility for Fun and Engagement

The following extension points were selected as possible extensions to make the game more engaging. It is a real challenge to discovering the right features which would make the game fun and engaging. The future decisions about such extension points depend on detailed usability studies with final users. Therefore, our concern was to enable some extensibility in aspects we find promising. Below are the possible extensions our architecture should be able to support without major disruptions in the overall design.

<u>Asteroid Conditions:</u>

- Change Gravity
- Include Atmospheric Events such as Wind
- <u>**How-to**</u>
    - Extend the PhysicsEngine.py and create new attributes in the Constants.py class (for wind, different value for gravity, etc.)

<u>Spaceship Types:</u>

- Let the user customize her Spaceship by defining attributes of cargo capacity (weight and volume), fuel capacity, speed, and landing system (e.g. parachutes, reverse throttle).
- <u>**How-to**</u>
    - Extend the PhysicsEngine.py and create new attributes in the Constants.py class (for wind, different value for gravity, etc

<u>Collective Prizes:</u>

- For each set of Mine Plots exploited the Base Station receives extra fuel. A set of Mine Plots correspond to a specific mineral, therefore, an Asteroid has as many Mine Field SETs as the different types of minerals.
- <u>**How-to**</u>
    - <u>**We did not support this extension**</u>. Major changes should be made in the methods in the GameServer (buyFuel, returnToBaseStation)

<u>New Types of Minerals:</u>

- Enable the addition of new types of minerals with different densities.
  - **<u>We did not support this extension</u>**. Major changes should be made in the methods of the GameServer (selectPlot) and in the SpaceshipViewer (mineral panel). All the game score and game goal calculations should also be modified.

2.5.2 Scalability

The game has no functional limitation of number of users. One possible technical limitation is the fact that we keep client states in the server side. So, an excess of clients must start hurting the server memory management at some point.

Besides that, the most process intensive requirement expected was the calculation of the Spaceship position during landing. The solution **designed and implemented** was to replicate in each client the engine responsible for such calculations. Hence, we will have neither network traffic nor server-side concurrency due to the calculation process.

2.6 How did we implement the Components and Connectors?

The client side is implemented as Model-View-Controller pattern. Each of the components and connectors are described below. Just below the figure, there is a legend with the names for each element. Each name is **traceable** to the Class Diagram available in Figure 3.

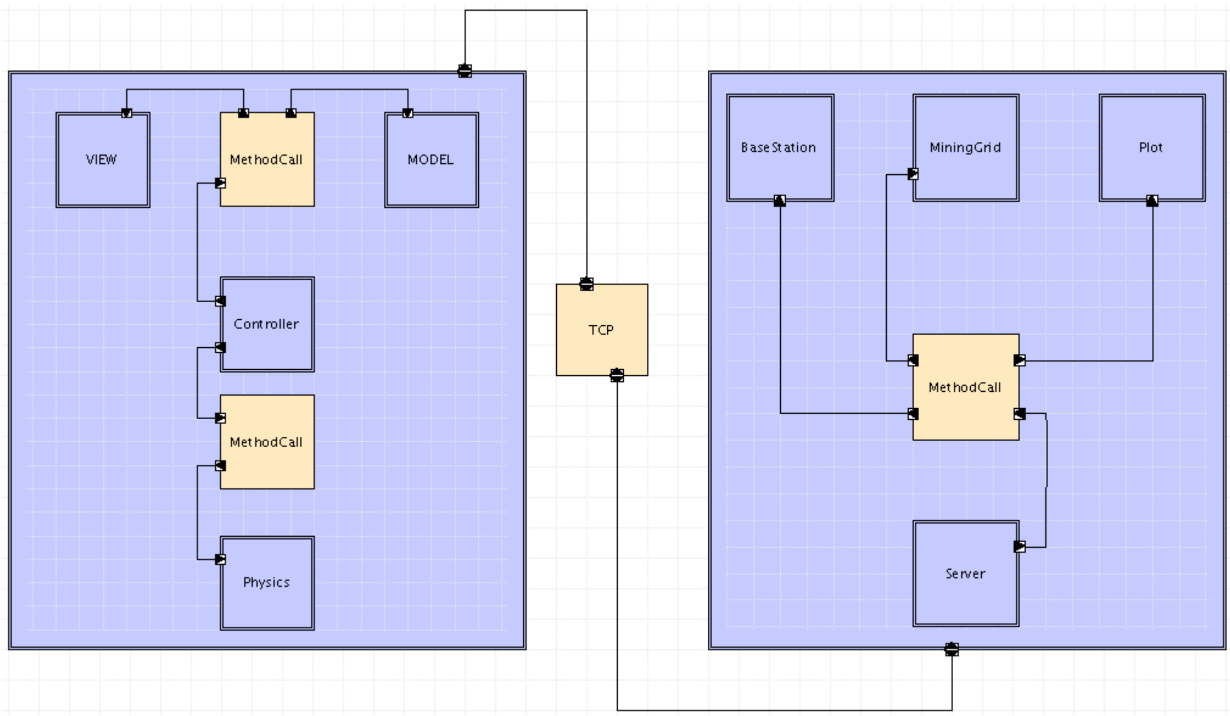


Figure 3 Architecture Diagram created in ArchStudio-4

Table 3 Legends and Traceability of Elements for Figure 3

Component Connector (xADL)	Description	Class Diagram (UML) Figure-3
MethodCall	Integrates via method calls	MethodCallConnector
TCP	Integrates via TCP calls	TCPConnector
Controller	Handles events generated by the GUI and updates the GUI with data retrieved from the GameServer	SpaceshipController
View	Manages the GUI	SpaceshipViewer
Server	GameServer keeps track of data shared among players	GameServer
Physics	Performs the calculations needed to correctly process the motion of the Spaceship in the screen.	PhysicsEngine

2.6.1 What changed in our Connector Model?

TCP connector (class TCPConnector inside GameServer.py)

Since we adopted PodSixNet,we had to change comply with the networking framework. The networking framework creates one instance of TCPConnector for each Server-Client pair. The reason for that is twofold. First, the framework discovers and keeps track of the client IP-Port addresses Second, the framework creates a representation of the client in the server side. I.e., the all the client states are also replicated in the server side.

Hence it affects the initial categorization we made for the TCPConnector.

In order to understand how we actually extended the framework, we had do create an extra UML diagram for that. See in <> how the client side (SpaceshipController) has to extend the framework ConnectionListener class. Likewise, at the server side, the GameServer had to extend a class. The TCPConnector also extends the framework Channel class.

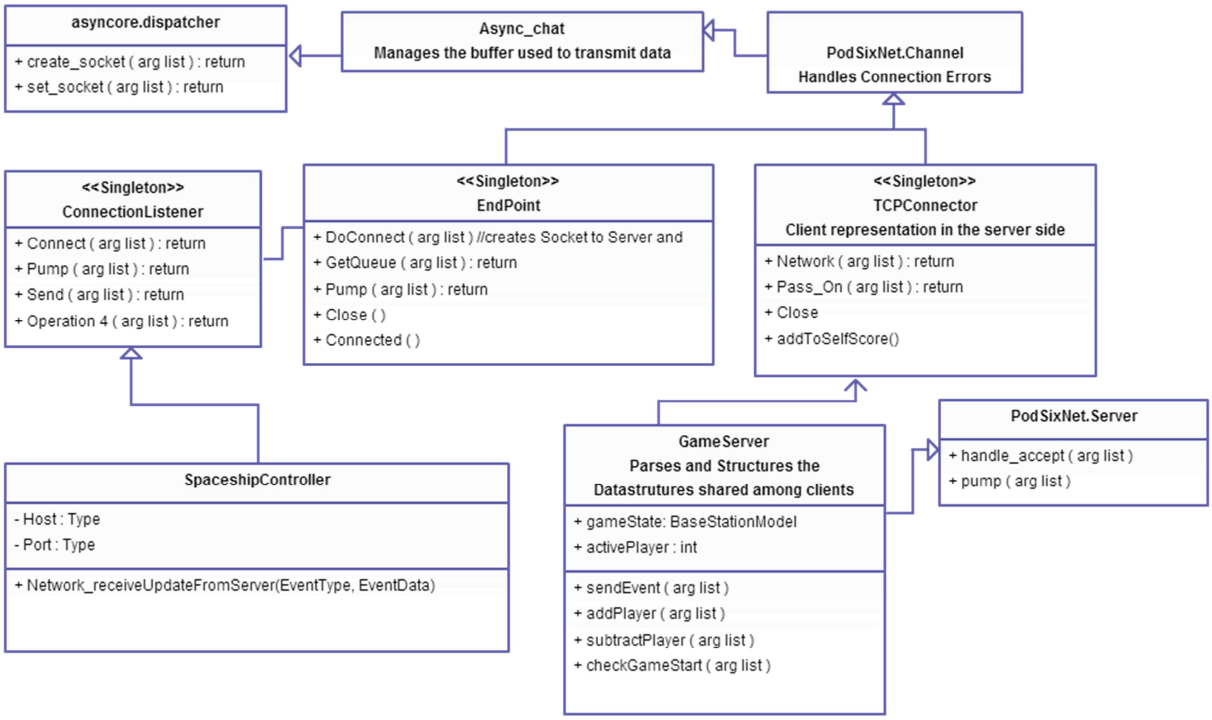


Figure 4 How we Implemented the TCPConnector

Role: Communication and **Statefull**

Type: Procedure Call Connector, Facilitator

Client and server both have an instance of this connector and they call the `send()` method of this connector to push data to the other side by calling a method network_<message-name>

Parameters: Host, Port, Data, Communication protocol. (default = 'TCP'), Network Adapter (default = 'podsixnet')

Podsixnet library is a lightweight network layer that asynchronously serializes network events and arbitrary data structures, which are delivered them via the TCPConnector. The TCP Connector in turn delivers the data by calling a method in the with an specific signature. The TCPConnector will be an instance of podsixnet Channel class and clearly act as a facilitator.

MethodCall Connectors

No changes were made in the design of the MethodCall Connectors. We used such connector types for all the communication between components residing in the same machine, such as:

- SpaceshipViewer and the PhysicsEngine
- TCPConnector and GameServer
- Client state representations in the server

Role: Communication

Type: Data Access Connectors

2.6.2 UML Class Diagram

The UML Diagram (Figure 5) maps directly to the classes we have in the source code. The few minor changes made were method signatures and a simplification of data structures in the BaseStationModel class. The simplification comprised the use of dictionary-like data structures (`{name:value, name:value}`), instead of classes.

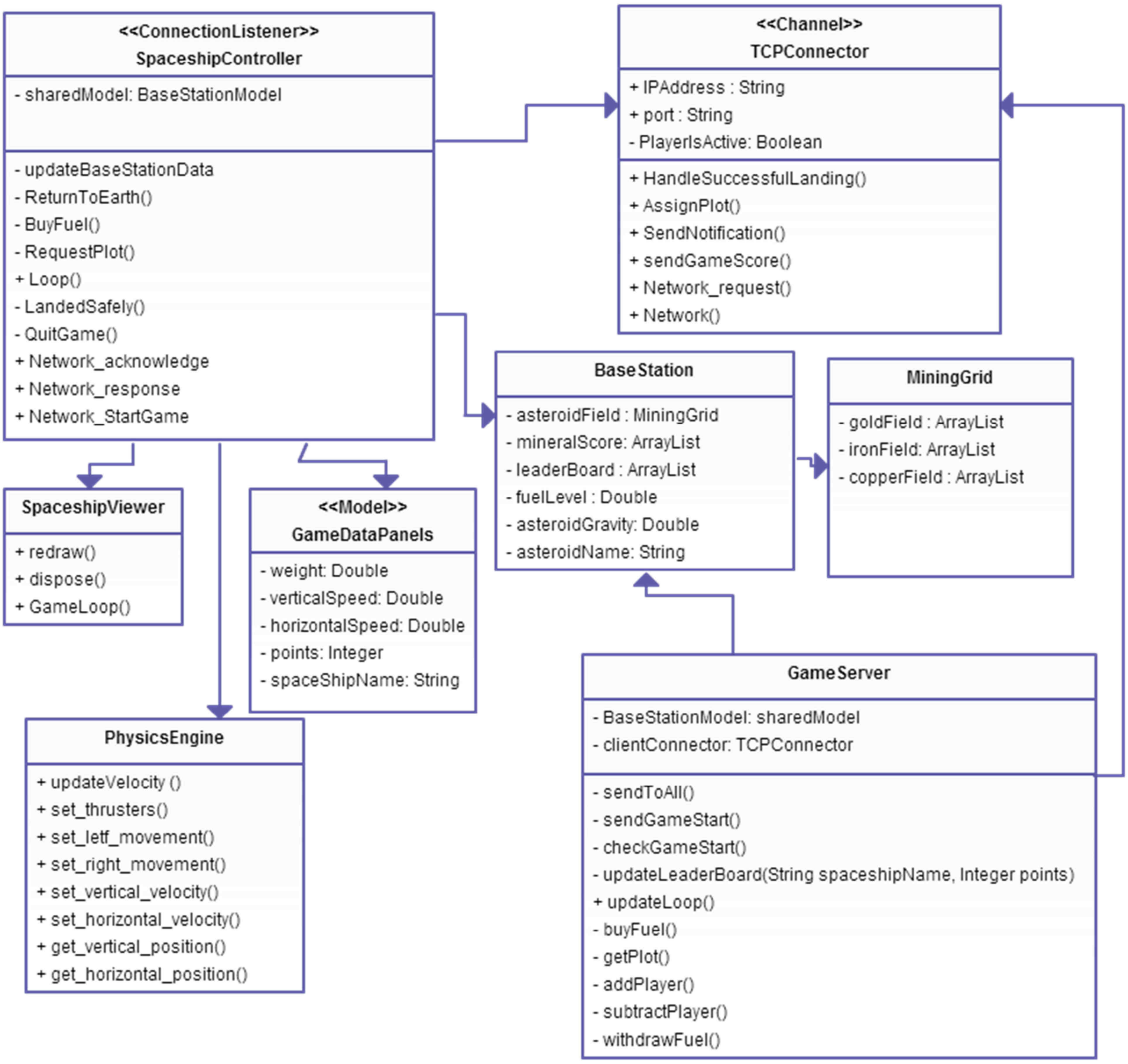


Figure 5 Final UML Class Diagram

2.6.3 UMLState Diagram

No changes were made in the UML State Diagram. Figure 6 depicts the state machine of the Spaceship. This simplicity of this state machine contrasts with the amount and complexity of events exchanged between the client and the server (shown in Table 4).

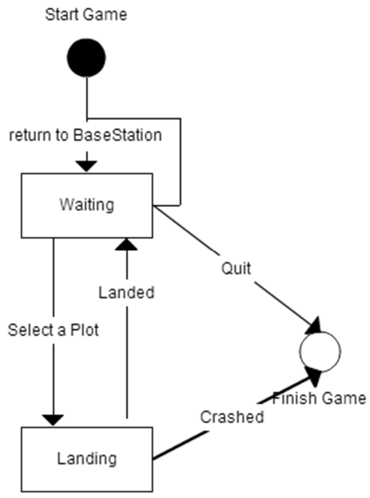


Figure 6 UML State Diagram with the Spaceship states

Below in Figure 7 are the two client interfaces waiting for the start event from the server.

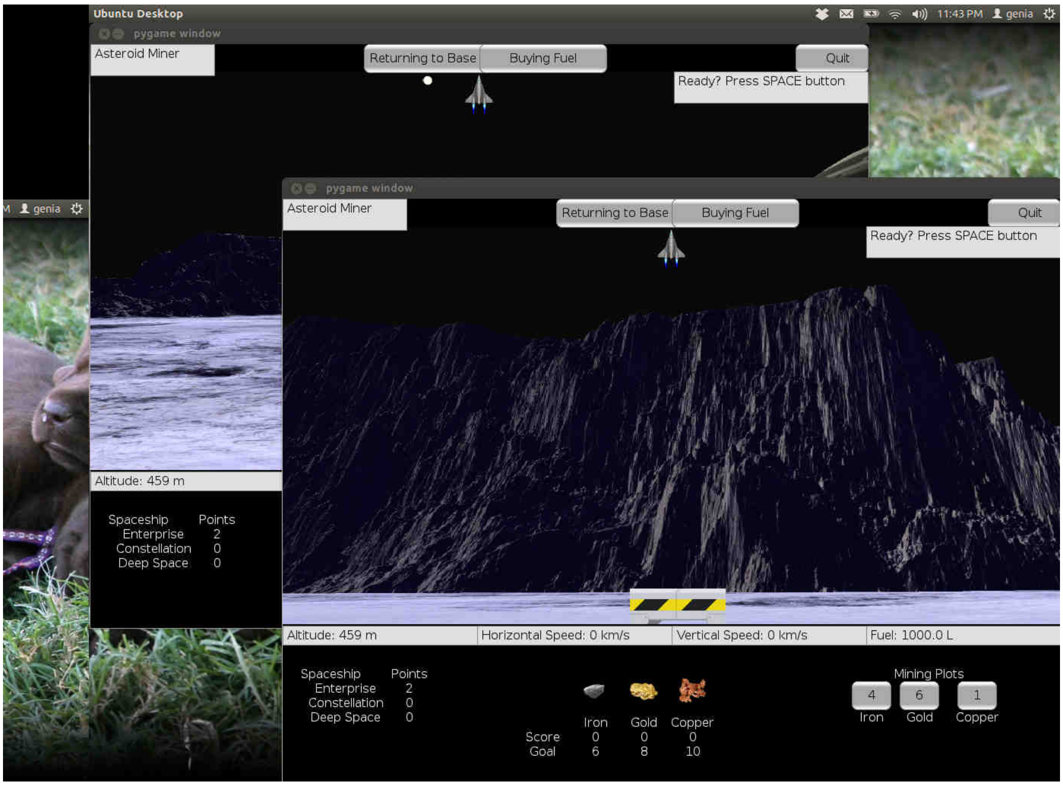


Figure 7 Clients Waiting for Game to Start (notification on top right corner)

2.6.4 Activities and Events Exchanged

No changes were made in the Activities Diagram.

Complementary to the Spaceship state machine, we also designed the events shared between the client and the server. For that we used a Graphical AADL Notation (Figure 8 and Figure 9) depict the sequence of such events in the context of the loops in the client side and in server side.

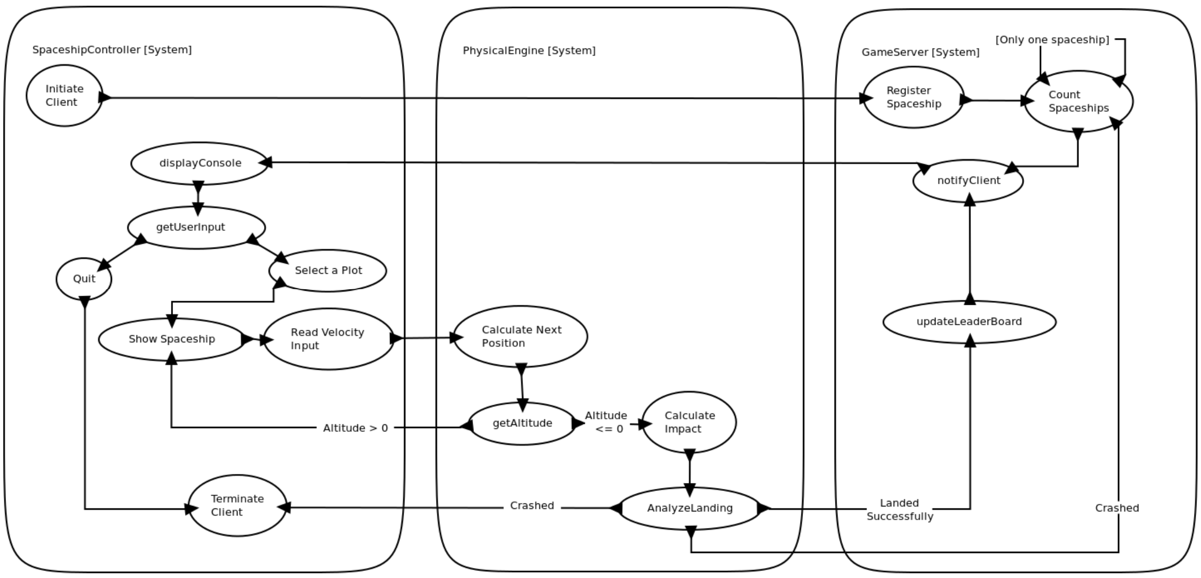


Figure 8 Landing Loop

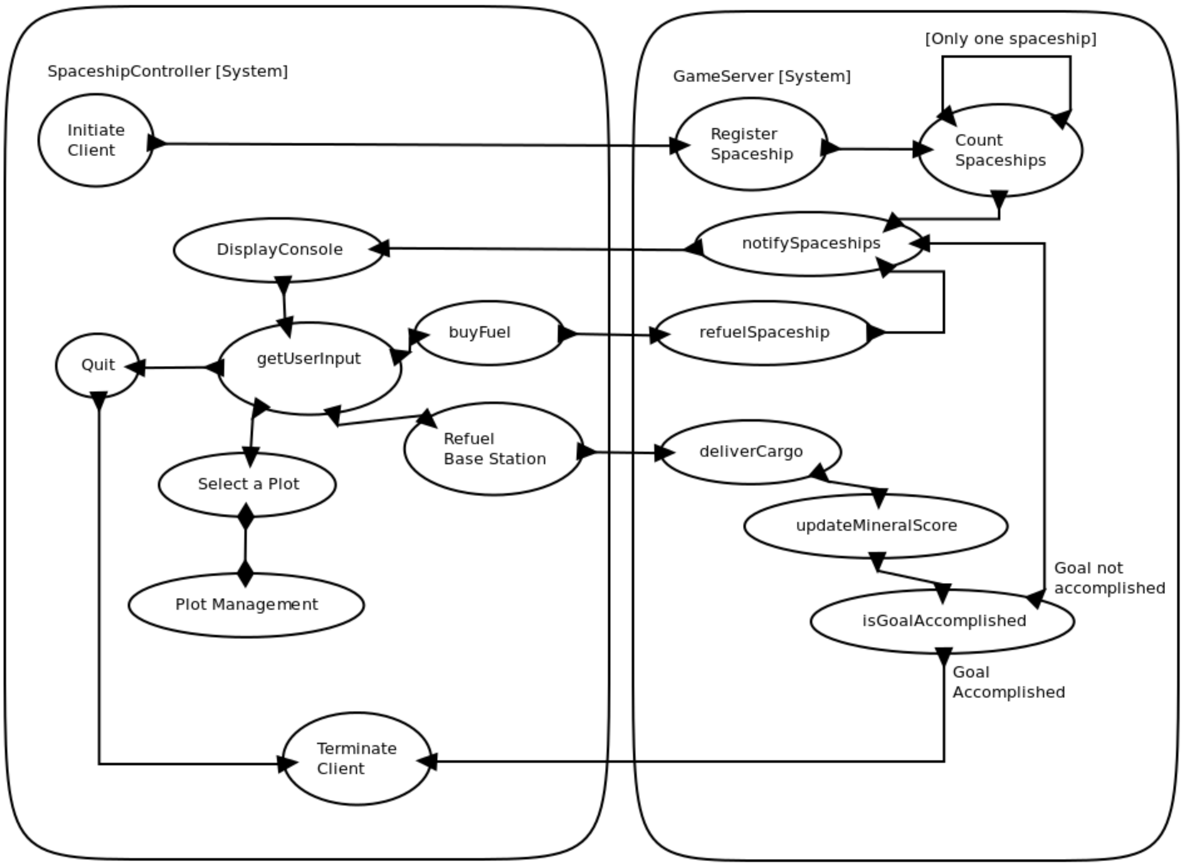


Figure 9 Server Loop

2.6.5 UMLSequence Diagrams

Regarding the Sequence Diagrams, we had two changes. First the method signatures changed. Second and most importantly, the **inversion of control** became clearer when we implemented. Client and Server classes are called by the PodSixNet framework, this was not explicit in the sequence diagrams.

In order to demonstrate how the methods in the class diagram generate the sequence of events describe in the AADL diagram, we created two UML Sequence Diagrams (Figure 10 and Figure 11).

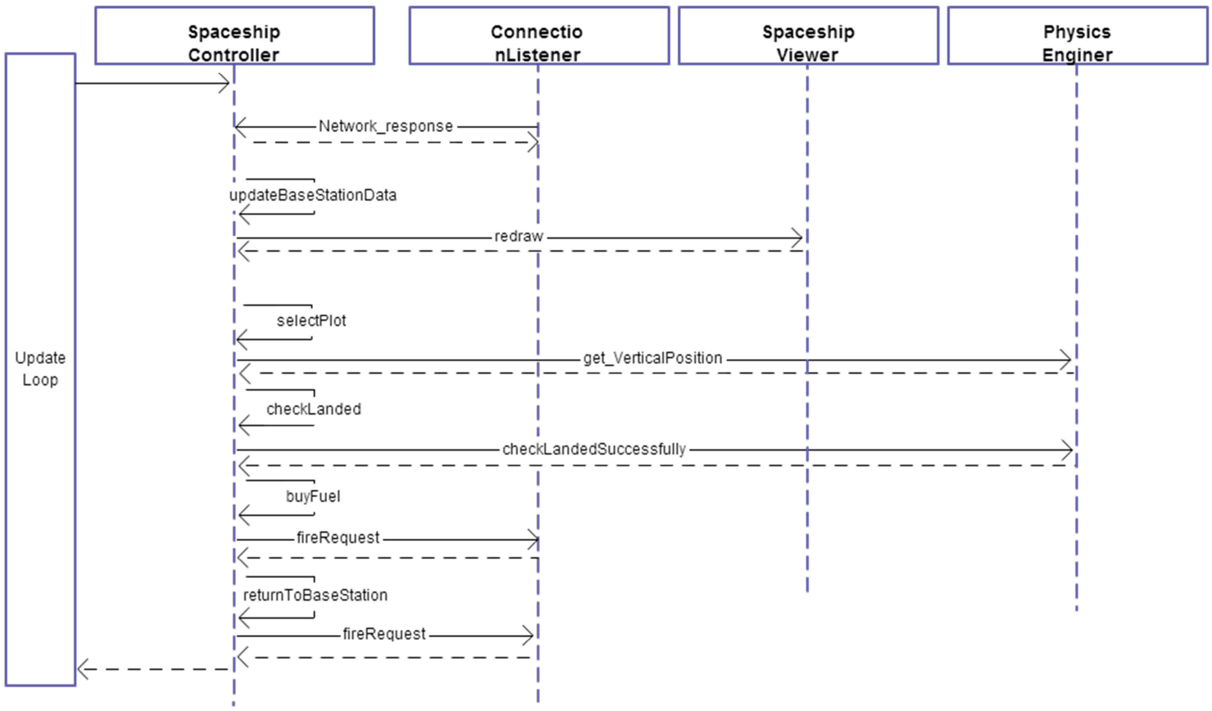


Figure 10 Sequence of method calls in the client side

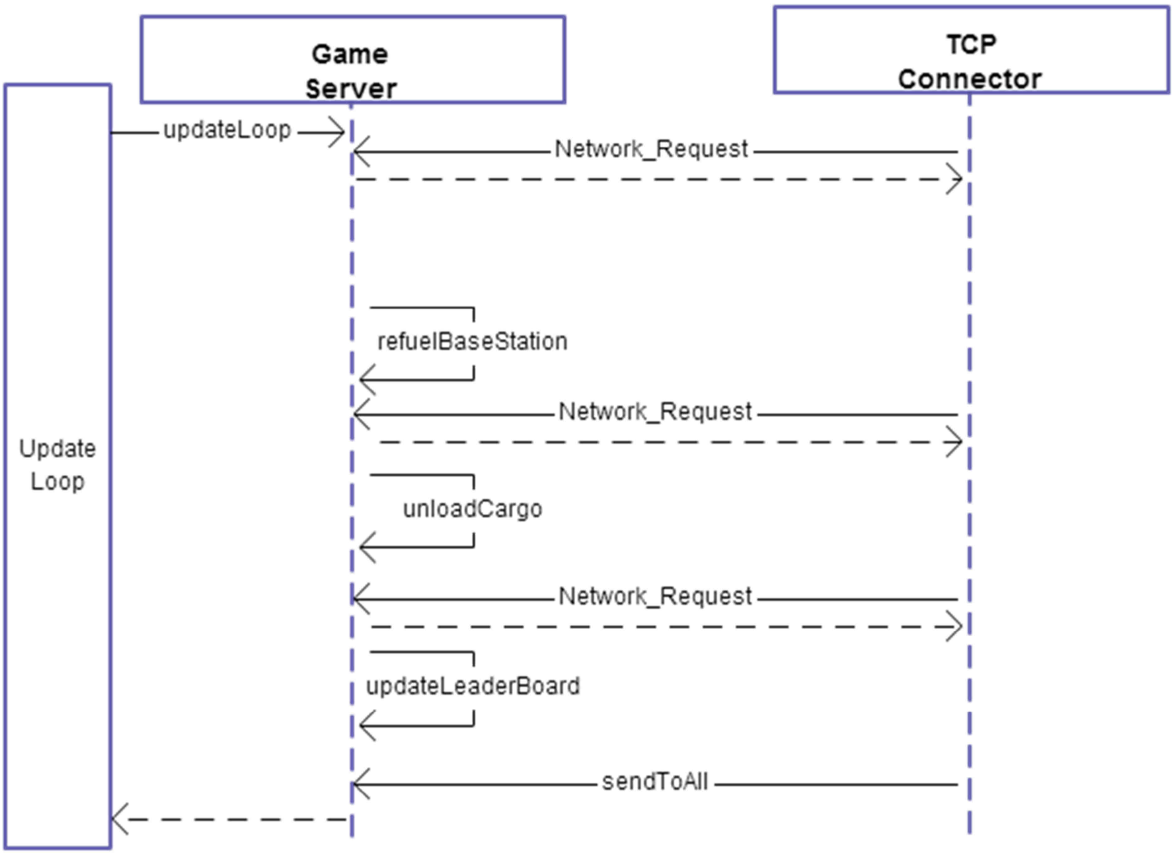


Figure 11 Sequence of method calls in the server side

2.7 How did we implement the Events and State Changes?

The fourth column in Table 4 maps the methods and classes responsible to handle each event. Some of our "state changes" trigger events while others do not. The reason is that some changes impact solely the client that generated the event. For example, the event of Spaceship landed impacts only the respective Spaceship client. In our game logic this information is not necessary for the other players. On the other hand, when a Spaceship requests to buy fuel for the BaseStation, this event affects the level of fuel seen by all the other players, hence a state change event must be raised. The following table describes all events we will have in our game.

Table 4 Game Events

Name of Event	Source	Trigger	Listener and Methods
Land Spaceship	Spaceship Viewer	LandButton = = clicked	SpaceshipController
Spaceship landed	Spaceship Viewer	(Altitude == 0.0) AND (VerticalMomentum <= 50.0) AND (HorizontalMomentum <= 20.0)	SpaceshipController Method: LandedSafely()
Spaceship crashed	Spaceship Viewer	(Altitude == 0.0) AND (VerticalMomentum > 50.0) OR (HorizontalMomentum > 20.0)	Spaceship Controller, LeaderBoard Method: CrashLanded()
Return to Earth	Spaceship Viewer	EarthButton = = clicked	Spaceship Controller, BaseStation Method:ReturnToEarth()
Buy fuel	Spaceship Viewer	BuyFuelButton = = clicked	GameServer Method: BuyFuel()
Choose plot to land	Spaceship Viewer	(PlotType = = available) AND (Plot Type clicked)	GameServer Method: RequestPlot
Quit game	Spaceship Viewer	QuitButton = = clicked	GameServer Method:QuitGame()
Game Start	GameServer	(NumberOfPlayers > = 2) AND (StartFlag <> 1)	SpaceshipController Method: Network_response()
Game Over	GameServer	(NumberOfPlayers < 2) AND (StartFlag == 1)	SpaceshipController Method: Network_response()
Base fuel changed	GameServer	FuelLevel.add(value) OR FuelLevel.subtract(value)	SpaceshipController Method: Network_response()
Leader Board changed	GameServer	leaderList.update(Spaceship, point)	SpaceshipController Method: Network_response()
MiningGrid changed	GameServer	grid.setFieldAvailable (latitude, longitude, 0)	SpaceshipController Method: Network_response()

3 Part-5 Assessment of our Experience

3.1 How hard was to maintain consistency?

Concerning consistency, we had to manually verify whether method names in the classes matched the activities and method calls in the sequence diagram. Further complicating the job, any later change implied in a huge impact on reviewing all the diagrams.

Therefore, it was very hard to keep consistency of the UML Class diagrams and Sequence diagrams. That was not the case for the State Diagrams and the XADL Activity Diagram. The problem we faced with UML is that it is a too low level model, which ends up having too tightly coupled mapping to the code. Hence, any minor change to the code renders the model inconsistent.

As we said in the previous report, in our opinion UML seems useful to rapidly sketch ideas and to plan ahead the implementation. We would neither use UML to validate/verify our model against our requirements nor to generate code from it. Since the effort to create an UML diagram is low, the cost of discarding it is also negligible.

3.2 How confident we are that consistency was maintained?

We are very confident in respect to the ADL and XADL diagrams, which were used respectively to model the component-connectors and the states-activities. On the other hand, we are by no means confident that the UML reflects 100% our code. If we were supposed to do that, we would end up having gigantic diagrams with several lines of method signatures.

3.3 What kind of changes were made in the model in the course of the implementation

We face three major changes related first to the choice and adaptation to the networking framework; second related to integrating the GUI to our game logic. Minor functional changes were already pointed in the previous sections. Below we describe the major changes.

3.3.1 Changing Astral Networking Library to PodSixNet

In our prescribed architecture, we initially proposed implementing multi-player networking using the Astral Networking Library. We quickly outstretched that plugin's functionality and documentation; in particular, documentation in numerous advanced features was simply not functioning. To implement our architecture, we instead used the PodSixNet networking adapter. It is stable and well-documented, and allowed us greater flexibility in implementing our client and server. This was a teachable moment for our team: the supposed "shortcut" of a framework such as Astral was actually more trouble than it was worth. Our time was better spent with the core plugin and we were able to carefully craft our system without the limitations of a higher-level framework. We were also able to learn how to troubleshoot multiplayer network interaction in Python.

3.3.2 Troubles with PodSixNet

The adaptation to PodSixNet were not larger because we had the time to study it thoroughly during design and modeling time. Otherwise, we are sure to have had major changes to our component-connector model. Two issues troubled us. First, is the communication truly both ways. In other words, both the client and the server are allowed to initiate a communication? The doubt stem from not initially reasoning how the server obtained the IP addresses of the clients, because we only provided as input the server IP address. Only after carefully reading the framework source code and digging three layers of classes (as demonstrated in our Class Diagram), we found the point where the framework obtains the necessary data. We had some hot group discussion concerning the effective need of this information, which in turn gave as a good clarification of the basics of remote procedure call architecture. Second issue involved the mapping between the events and the handing methods. We did not realize in the beginning (even after having some code running) that the framework unmarshalls an specific message text passed in the event and composes a call to a method. I.e., the name of the method is composition of the words "Network_" and the text passed in the event (e.g., "response"). After grasping that, we could create new methods instead of relying on single one letting it make all unfolding calls based another data passed in the event.

3.3.3 GUI Integration Problems

We used *pygame*, a Python gaming library, to build our game console and mining features. A *pygame* application usually includes UI initialization and a game-update loop where components of the game are drawn to the *pygame* canvas. One interesting aspect of our UI is that we exchange more than lander coordinates across the network. We chose *pgu*, a GUI library for *python*, to implement our Client-Side view. One advantage of this library is that it provided a framework for us to add common UI elements such as dialogs, tables, and buttons easily.

However, the library has not been updated since 2011, and the documentation varies in quality. We had to work around certain connector bugs with native event handling, but this was a manageable task. We realized that these bugs in event handling existed later on in our development process, so the logical choice was to work around them and not start from scratch. It is likely that many software development teams encounter these decisions - at what point is the architecture supporting too many legacy versions? Too many quick-fix hacks? Every decision is expensive when you stray from your planned architecture.

3.4 How much the architectural model helped during the implementation?

3.4.1 Because Important Complexity was Pruned during Modeling

Modeling forced us to think about how various components and connectors would interact and also gave us some insight about which properties our system would exhibit. For example, initially we thought of using a database to maintain the game state, but later understanding that the states could be shared at the server side, we confidently dropped the database idea. This also implied in a compromise. On one hand, we would have lesser complexity by stripping out a persistent data access layer and all its database connectors. On the other hand, we would decrease scalability, since the server will now be keeping the complete game state in memory. Hence, our system would be at closer limit to the maximum number of players we could host.

3.4.2 Because Non-Functional Requirements Were Realistically Set during Modeling

Another similar important decision was sharing game data and not the visual space among the different players. This decision does not have any impact on the architecture of the system but it saved us a lot of accidental complexity related to drawing objects of one player on the screens of all the other players. Moreover, we realized that such operations would require a lot of synchronization triggered by every movement made by one player and the respective and necessary propagation to all other players. Such would end up in slowing down the user experience.

3.4.3 But Some Things Required Testing to Reason About

While testing our system, we found ourselves reflecting on many design decisions that we had not realized would be challenging when we initially designed our architecture. We realized that player experience - the experience of a user from network connection to finishing the program - was something that we should have considered more deeply and perhaps even storyboarded in great detail step by step. This realization came about as a result of the challenge of testing for different GUI functionality. Even with modular network architecture, GUI testing is a slow and painstaking process. No system like Selenium (http://docs.seleniumhq.org/, an automated test framework for browser applications) exists for game applications.

3.5 If we had to do it again, what would we have done differently?

3.5.1 Granularity Level of the Model

Concerning the level of detail provided by the UML diagrams, we believe they did not pay off the cost of creating and maintaining them. Many other classes and methods appeared which were not represented in the UML class diagram. The same is true for the UML Sequence diagram.

The big take away for us is to learn how to master the modeling in the right level of abstraction while we still are able to perform proofs of technology with the candidate frameworks. To make sense of framework functioning and how it affects our initial model, some level of detail above lines and boxes is needed. The solution, we suggest, is to rely more on the taxonomy of connectors.

3.5.2 Performing Proofs of Technology during Design and Modeling

The architectural uncertainty and trouble we faced with the frameworks for GUI and for Networking could have been avoided if we had implemented small prototypes. Such would have given us a better understanding of the how those would have affected our models.