Space Traders Group Project Report

Peter Collingbourne, Jiefei Ma, Steven Lovegrove

June 6, 2005

1 Introduction

1.1 Requirements and Targets

The specification for this project called for a "multi-user space trading game to be played on the web", which uses a database for storage of game data. The game must be implemented in a language that supports CGI (such as Perl or PHP), Java or ASP on the server side, such that no unusual requirements are placed on the web server. The game must operate in one of Mozilla, Firefox or Internet Explorer on a departmental lab machine. The database must be one of PostgreSQL or SQL Server.

We additionally laid down the following targets for our game:

- The game must be real-time so that users are not kept waiting for other users. This entails that game data must be continuously updated on screen.
- The game must include interactive graphical displays in order to interest the user.
- The game should allow for interaction between users via means of private mail, chat, trading and space battles.

1.2 Rules

Our Space Trader game is based around a galaxy in which users can trade gold, food, fuel and iron. The aim of the game is to survive longer than all of the other players in the game, and make best use of the resources the galaxy has to offer. Players may own any number of Planets: centres for trading, and where the planet owner can build facilities for extracting resources, or for building new ships, and Convoys: fleets of ships that can move freely around the galaxy. A Player loses and is unable to play the game if they have no planets or convoys.

Convoys can hold food, fuel and iron and can meet with each other at planets in order to facilitate trading. Ships have a speed and a capacity. The speed of a convoy is the maximum speed of its slowest ship. The capacity of a convoy is the total capacity of all its ships, and dictates the resources a convoy is able to carry. A convoy is only able to travel to a planet if it has enough fuel. A convoy is destroyed if it runs out of fuel or food.

Planets are clustered around stars in solar-systems. The real-time nature of the game, means that the time taken to travel between two planets is proportional to the distance between them, and the speed of the convoy. Interstellar travel will take much longer than between planets in the same solar-system.

When a player's convoy is at one of their planets, their resources can be exchanged freely up to the maximum capacity of the convoy. Players can build facilities such as Mines to produce Iron, Farms to produce food, Refineries to produce fuel, and Factories to produce ships on a planet, if that planet holds enough resources to do so. After some time, a facility will get run-down, and be demolished.

Factories are able to produce many different kinds of ships which take a specified number of resources and time to build. Building of ships may be queued, where building of the next ship takes place after the first finishes. Payment for a ship is taken immediately before it is produced. If there are insufficient resources, no ship in the queue is produced until the resources are deposited or produced at the planet.

Ships produced by a factory, when complete, will be located at that planet in a convoy containing only itself. Convoys may be merged to form a larger convoy containing the ships in each, or split to create smaller convoys. Convoys can also be given descriptive names for purposes of identification.

Two players can trade with each other when they have Convoys located on the same planet. If one player requests to trade with a second player, the second player is notified, and is able to begin negotiation. Resources are exchanged only if both players accept the conditions. Private chat is available for this transaction.

Public chat is available in the game to arrange rendezvous's, and to allow the galaxy's community to converse, brag or deceive, as well as private mail, which can be sent to online or offline users.

1.3 Structural Overview

This section gives an overview of the overall structure of the client and server sides of the program.

As can be seen from Figure 1, there are two main ways for the client to communicate with the server: via a web browser or via a client side applet. The web browser is used in order to register and log into the game, and uses HTTP [2] to download the applet to the user's machine. The applet implements the main part of the game: it is the main interface by which users play the game.

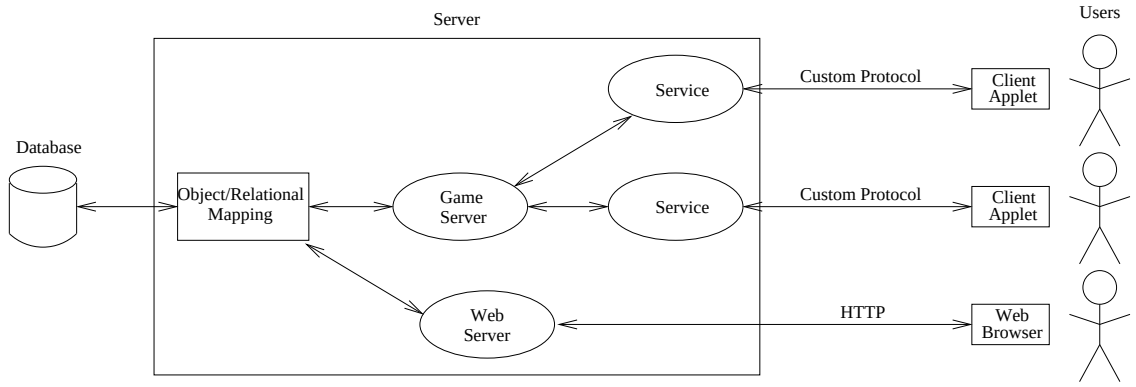

Figure 1: Architectural Overview

```
C: (LogIn "peter" "secret")
S: (LoggedIn :id 13 :name "peter")
S: (OnlineList (13))
S: (NotifyMessageCount 14)
S: (UpdateResources :iron 8877 :fuel 8688 :food 9063 :gold 1799)
C: (GetOwnConvoys)
S: (GotOwnConvoys (:size 1 :id 30))
C: (GetConvoy 30)
S:      (GotConvoy :ships ((:capacity 100 :speed 10 :description "Can do the Kessel run
in less than 12 parsecs" :name "Millennium Falcon" :id 31)) :iron 6 :fuel 5 :food 4
:speed 10 :size 1 :star 8 :planet 10)
```

Figure 2: A sample session between server and client

In order to facilitate communication between the server and client, a custom protocol using a message based system is employed. The protocol is text-based and operates over a TCP connection. It is also easy for humans to construct messages in this protocol, making both the server and the client easier to debug.

The protocol uses a Lisp-like syntax in order to communicate with the server. This protocol is described in more detail in Section 4.2. Generally a 'conversation' between two parties using the protocol will involve the client informing the server of the actions performed by the user and the server informing the client of any relevant changes in the system state. Figure 2 shows a sample session between the client and server in which the user logs in and retrieves information about one of their convoys.

2 The Client Side

The client side user interface was implemented as a Java applet. This afforded us a greater level of interactivity as a user interface could be implemented without the need to make extraneous requests to a server, as required by a pure web-based interface. It also allows convenience for the user, as they can play the game without having to specifically download and install a separate executable.

The interface is implemented by Swing, Java's modern API for graphical user interfaces. The decision to use Swing was made as it provides a consistent user interface on several platforms and it provided several graphical 'widgets' required to implement the interface. The features Swing provides, such as the aforementioned widgets and layout managers, allowed us to rapidly and easily write a user interface.

When the applet initialises it opens a window in which all user interaction takes place. It was decided that interaction would occur in this manner to parallel the interface provided by contemporary PC and console games. The user interface is internally divided into screens and subscreens. Every screen that is accessible via the applet may have a subscreen which is easily changeable. The applet maintains a stack of screens in order to mirror the easily programmable paradigm of pop-up windows – when a particular screen wishes to relinquish control to the previous screen it can simply 'pop' itself off the stack.

Screens and subscreens play an important role in the handling of incoming messages. Each screen may register a number of commands with the game client. When an incoming message is received, the function name is matched against the topmost screen on the stack that can handle the message. This enables us to receive notification messages and update the user interface for screen elements which are currently invisible (i.e. handled by a screen which is not at the top of the stack), thus simplifying the code. Subscreens may also specify commands, thus clarifying the subscreen-dependent and subscreen-independent code. Figure 3 gives a UML state chart which illustrates the possible transitions between screens and subscreens.

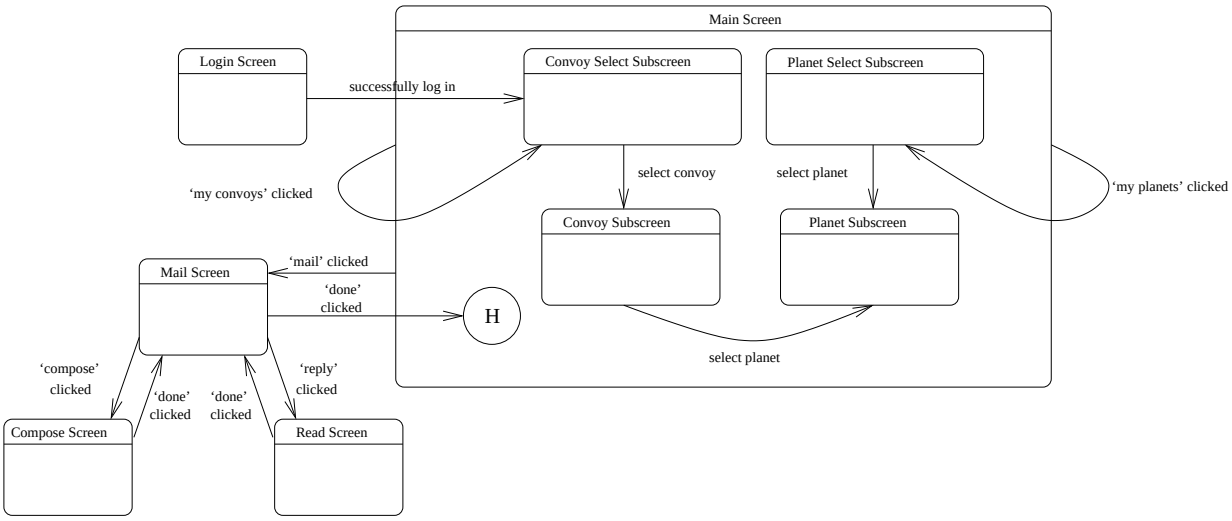

Figure 3: Client State Chart

2.1 The Main Screen

The main screen, in which the user spends most of their time, has been designed such that the user has easy access to the most important features – they can read mail, read broadcast messages, examine the list of online players and access a list of their convoys and planets allowing the user to manage them easily. Figure 4 illustrates the features of the main screen.

The main screen can accommodate subscreens in its centre – some of the subscreens used by the main screen are described below.

2.1.1 The Convoy Selection Subscreen

This subscreen is accessible via the 'My Convoys' button on the main screen and allows the user to select a convoy to manage by clicking the convoy with the mouse. Each convoy is labelled with its name, the number of ships in the convoy and its destination to allow the user to easily select their desired convoy. Users may also merge and split convoys at this subscreen by means of the **Merge** and **Split** buttons located at the bottom of the subscreen.

2.1.2 The Planet Selection Subscreen

This subscreen is accessible via the 'My Planets' button on the main screen and presents the list of planets that the user owns. Users may click on a planet in order to view information on the planet via the planet subscreen, described in Section 2.1.4.

2.1.3 The Convoy Subscreen

This subscreen is accessed if the user clicks on one of their convoys in the convoy selection subscreen (section 2.1.1). It allows a user to view and manipulate their convoys. It consists of four tabs:

- The *convoy* tab displays general information about the convoy such as its name, maximum speed and the ships it contains. It also allows the user to change the name of the convoy.
- The *view* tab shows an abstract view of the current position of the convoy. It may either show a rotatable 3D map of the planets in the galaxy it is currently in, or if the convoy is moving it displays a moving starfield in order to convey the sensation of movement. Selecting a planet in the 3D map takes the user to the planet subscreen for that particular planet (Section 2.1.4).
- The *resources* tab shows the current level of resources held in the convoy.
- The *travel* tab allows the user to move their convoy. It displays a list of galaxies in the left hand column and a list of planets in the right hand column. To move to a particular planet the user selects the galaxy and the planet and presses 'Move to Planet'. The user is then taken to the View tab where they can view the time progress of their journey.

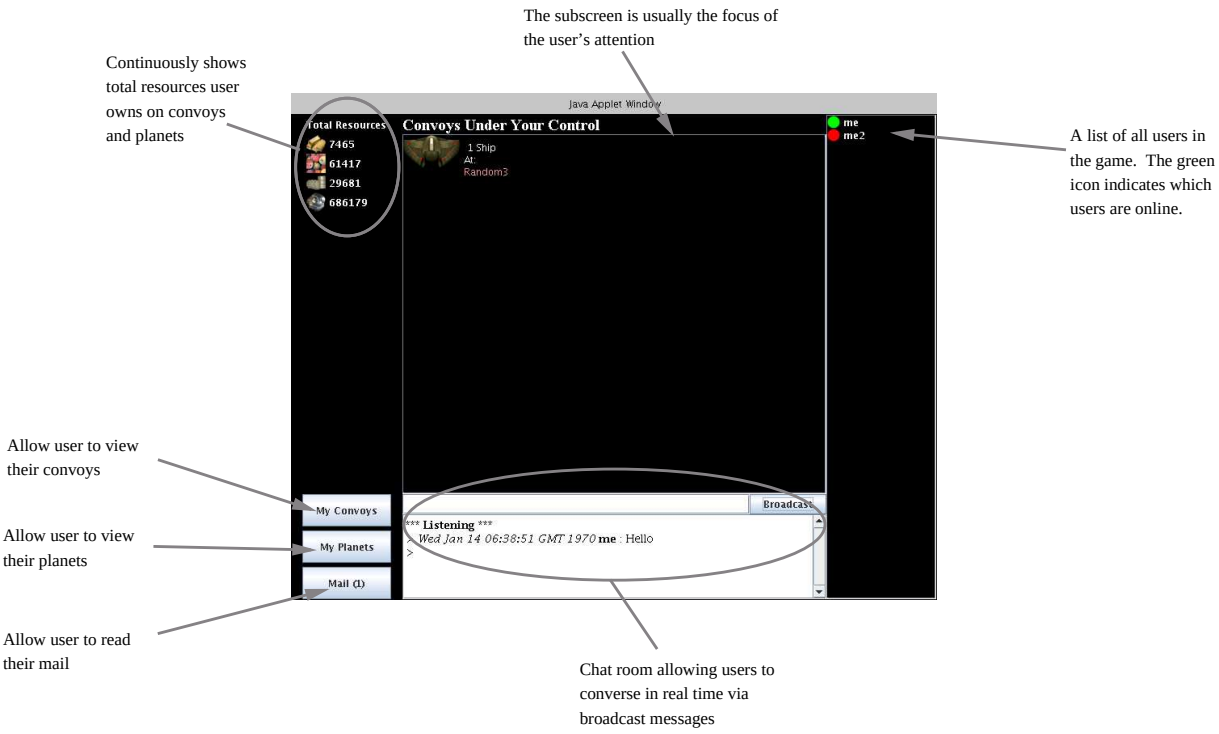

Figure 4: Main Screen Feature Diagram

2.1.4 The Planet Subscreen

The planet subscreen gives the user information about a planet and allows them to trade with other users on that planet and transfer goods from the planet. It is composed of five tabs:

- The *planet* tab gives the planet name and owner. If there are no facilities built on the planet it also gives the user the option of taking ownership of the planet via the 'Take Ownership' button.
- The *facilities* tab allows the user to view a list of facilities on the planet. If the user owns the planet they may build facilities by selecting the type of facility and selecting the 'Build' button.
- The *resources* tab displays the current level of resources on the planet.
- The *trade* tab allows the user to trade with other users currently on the planet. Trading is described in more detail in section 2.3.
- The *stock* tab allows the user to store resources (i.e. food, fuel and iron) onto the planet or stock goods produced by the planet onto the convoy (ships), given that both the planet and the convoy belong to the player.

2.2 The Mail Screens

These screens allow the user to read and compose mail. Mail is private and may be sent from one user to another. There are three screens involved in reading mail:

- The *mail screen* displays the user's inbox in the form of a traditional mail client. Each message indicates the sender, the date of receipt, the subject and whether or not the message has been read. Selecting a mail and pressing the **Read** button allows the user to read it. The **Compose** button allows the user to compose a message. The **Back** button takes the user back to the main screen. This screen responds instantly to changes in the user inbox via messages from the server.
- The *mail read screen* allows the user to read a particular message. It consists of text fields showing the sender, date, subject and body of the message, as well as two buttons: **Reply** allowing the user to reply to the message and **Done** which takes the user back to the mail screen.

- The *mail compose screen* allows the user to compose a message. This screen may be reached either by composing a new message or by replying to an existing message. There are three text fields which the user must fill in: the recipient, the subject and the body. The **Send** button allows the user to send the message, and the **Cancel** button takes them back to the previous screen.

2.3 Trading

Players can trade with each other for the properties they own. These properties include gold, food, fuel and iron. In order to let the players trade fairly, a set of requirements has to be met and a sequence of actions has to be performed before a deal is made successfully between two players.
Requirements:

1. Both of the players (and their convoys) have to be on the planet.
2. Both of the players agree to trade.

Actions:

1. Player **A** selects another player **B** from a potential traders list and sends a *request* for trade. Then **A** has to wait until **B** *accepts* or *rejects* the request. In the mean time, **A** may *cancel* the request. Once **B** accepts the request, then the negotiation between **A** and **B** starts.
2. The offer made by the opponent will be displayed on the current player's negotiation screen. Also, the two players can bargain over the offers by using the *Instant Message* like function at the bottom of the negotiation screen.
3. The current player may modify their offer by changing the values of goods in the corresponding `Spinner`s, and/or *accept* the current offer by pressing the **Accept** button. After sending out the acceptance, the current player has to wait for the opponent to *accept* the offer.
4. Changes to the offer by the current player will be updated on the opponent's negotiation screen instantly. In order to prevent players from cheating, any changes made to the offer will cause the cancellation of acceptance of the offer on both sides. Therefore, if a player has previously accepted the old offer, she has to *accept* it again.
5. A deal is made only when both players have accepted the offer.

3 The Server Side

The server consists of two parts: a game server and a web server. The web server responds to requests from the user's web browser and the game server communicates with the client applet using the custom protocol. These two parts run in the same process in order to provide the user with an integrated user interface, as will be described. Both the game server and web server were written in Java. The decision to use Java was made for the following reasons:

- All the members of the group were familiar with this language;
- The client was also written in Java, so writing the server in Java would allow us to reuse code;
- Java offered us a pre-made embedded web server via the use of servlets, so writing the server in Java decreases any development time on a web server.

3.1 Game Server

The game server operates by responding accordingly to any incoming messages from the client, as well as informing the client about any relevant changes in the system state. The game server has a wide range of responsibilities, chief among these being:

1. Allowing the user to log in. When the server receives a message requesting a login, the username and password are checked against the database. If they are correct, the user is logged in, otherwise they are rejected.
2. Informing the client of any change as to which users are currently logged in. This ensures that the list of online players in the client is always correct.
3. Ensuring that broadcast messages are broadcast to every logged in player. This enables a real-time chat facility.
4. Informing the client whenever new mail is received. This allows the client to update the message count, and the user's inbox if it is visible, at any time.
5. Allowing the client to retrieve information about stars, convoys, planets and facilities. This allows the client to present the information to the user. The server will also inform clients about changes to this information provided the client is viewing the correct screen.
6. Facilitating trading between users. This involves informing the other party whenever a trade request is made, whether the other party accepts the request, whenever the amount offered by the other party changes, whenever the other party accepts the trade and finally when the trade is complete.

```
Session sess = HibernateUtil.currentSession();
Transaction tx = sess.beginTransaction();
...
Convoy convoy = (Convoy)sess.load(Convoy.class, new Integer(convoyId));
convoy.setName(newName);
sess.save(convoy);
...
sess.flush();
tx.commit();
HibernateUtil.closeSession();
```

Figure 5: Example code of using **Hibernate**

3.2 Web Server

An important part of the server is the web interface. This serves as the 'launchpad' to the game by allowing the user to log in, and transferring the applet to the user via HTTP. The web server was implemented by Apache Tomcat 5.5.9 at the time of implementation, but should be easily portable to other versions of Tomcat. It invokes the Tomcat server in embedded mode, such that it runs in the same process space as the game server and affords a greater level of control over the operations of the Tomcat server. For example, accessing the applet via the web interface allows the user to log in via the web and start the game without logging in again in the applet. This is achieved by using a custom page to invoke the applet, specifying the session ID as one of the `param` tags. The applet will pass the session ID as one of the parameters of a command to the server. When the game server receives this session ID, it can ask the web server which logged in user corresponds to the session ID in order to authenticate them. This is one of the advantages of running the game server in the same process as the web server.

3.3 Object/Relational Mapping

In the development of our project, we used the `org.hibernate` package. **Hibernate** [1] is a powerful, ultra-high performance object/relational persistence and query service for Java.

From time to time, the server needs to interact with the database in order to save and retrieve information about the running game. Using only native SQL queries is tedious and error-prone due to the object oriented nature of the game. In contrast, the *Hibernate Query Language*, designed as a minimal object oriented extension to SQL, provides an elegant bridge between the object and relational worlds. Thanks to Hibernate, we could develop the game as a set of persistent classes following common Java idioms (including *association*, *inheritance*, *polymorphism*, *composition* and the *Java collections framework*), and express queries using native SQL or Java-based Criteria and Example queries.

The `hibernate` package allows objects to persist beyond the lifetime of the game server application by saving and restoring objects into a database. In this respect, database data integrity and maintenance is provided transparently by Hibernate.

4 Package Information

The project, being implemented in Java on both sides, was split into packages in order to attempt to enforce a clean separation of duties between distinct portions of the project. The following subsections describe some of the packages which make up the system.

4.1 The `galaxy` Package

The `galaxy` package is responsible for maintaining the game state, and defines how objects in the galaxy are able to interact with one another. The structure of the package (as can be seen in Figure 6) is designed to minimise coupling, and maximise cohesion, to simplify use and implementation of the package. The `GameServer` uses the `galaxy` package in conjunction with Hibernate to store state in a database.

The key classes in the package are described below.

Game The root class in the galaxy structure that represents the state of the game. This class is responsible for two main collections: the collection of `Player`s, and the Collection of `Star`s. The `Game` class also has *static* methods that initialise the Facilities `ProductPool`, discussed later. This class also has the public method `tick()` which is how the package processes time. The `tick()` method invokes `tick()` for the `Game`s `Player`s and `Star`s, being cascaded down to classes that need some concept of time. The `tick()` interval is decided by the `GameServer`, and this implementation reduces server load, and simplifies computation by making time discrete.

Star This class provides a way of grouping planets, and makes it easier for a user to distinguish between planets which may be in different Solar-Systems. Functionally, this class provides a method for iterating over its collection of `Planet`s.

Planet This class maintains a collection of `Facility`s, its `Resources` and its owner. The `Planet`'s owner can be `null` representing it has no owner, or refer to a `Player`. The class's `Resources` attribute refers to the food, fuel and iron that are stored on the `Planet`.

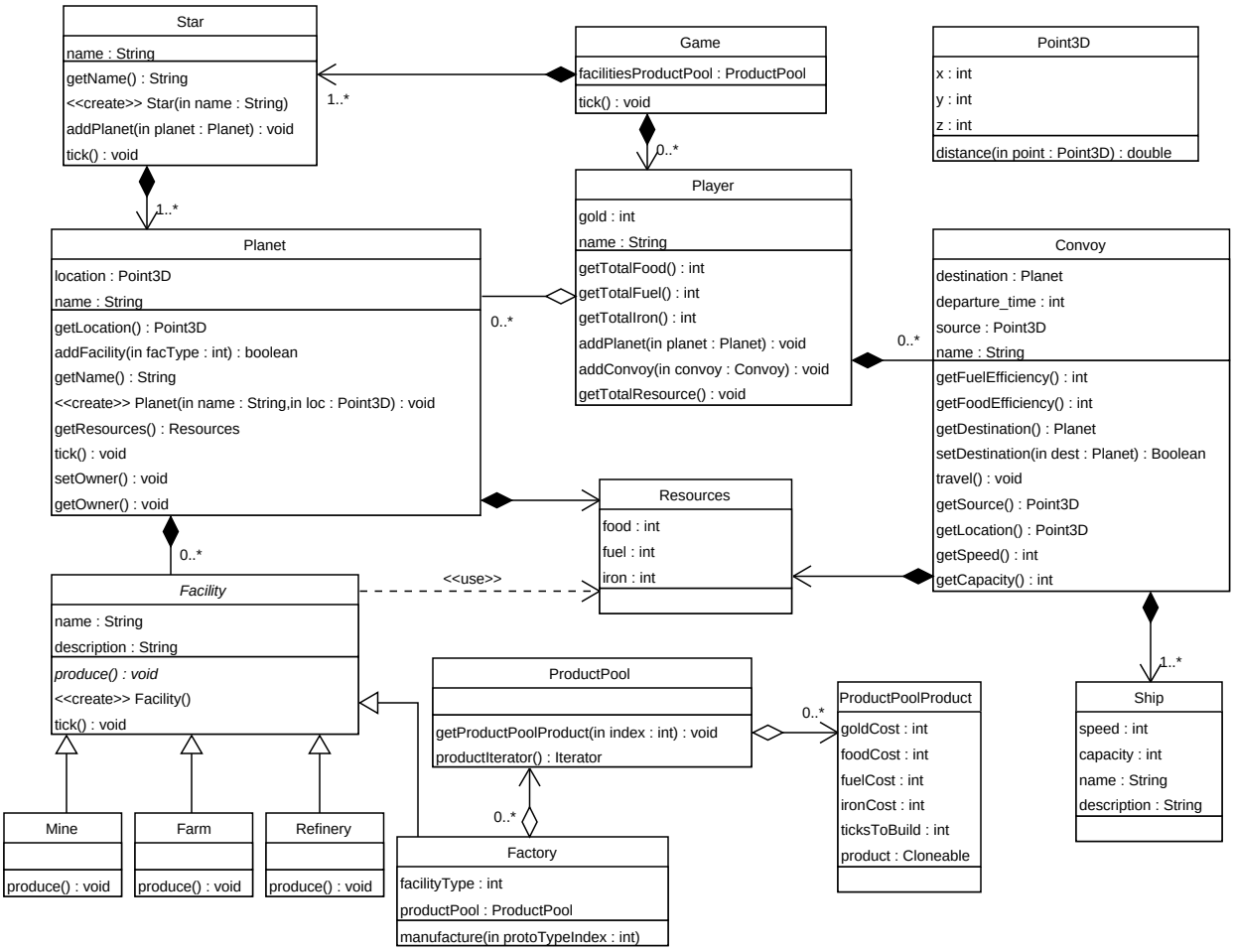

Figure 6: *galaxy* package class structure diagram

`Convoy`s belonging to a `Planet`'s owner can 'stock' `Resources`, and produce of the `Planet`'s `Facility`s are added to the `Planet`'s `Resources`.

Player This class represents the real-world Player and their resources in the game. It holds two main collections, that of `Planet`s owned, and `Convoy`s owned. It also stores the player's *gold*, which is a resource that doesn't reside in any location, only with the player. The class also has methods used for establishing the player's total resources; those resources belonging to the players `Convoy`'s and `Planet`'s,

Convoy This class represents a group of ships that are controlled as one logical block. A `Convoy` has a `destination`, and a `source` of type `Point3D`, and a `departureTime`. The class also has methods for calculating the `Convoy`'s overall speed and capacity based on the `Ship`s it contains. From these attributes, the `Convoy`'s location at any time can be calculated. A `Convoy` is defined to be stationary if its `source` and `destination` are the same, or the current-time - departureTime is greater than the time it would take to travel between the source and destination at the minimum of the speeds of the `Ship`s that the `Convoy` contains. The class contains a method `travel( Planet dest )` which will allow subsequent calls to `getLocation()` to provide the ship's exact location in space.

Ship This class describes the properties of a space-ship, namely its speed, capacity, descriptive name, and description. `Ship`s are created using the *prototype design pattern* from a `ProductPool` within a factory, and as such, to enable copying the object, implement `Cloneable`. See `ProductPool` and `ProductPoolProduct` for more details.

Facility This is an Abstract class representing objects that can be built on a `Planet`. The class holds a reference to its parent `Planet`, and defines the Abstract method `produce()`. The `Facility` class also has the method `tick()`, whose implementation is simply to invoke `produce()`. The idea is that a sub-class of a `Facility` will have some output with respect to time, and alter the state of the

game in some way. `Facility`s are created using the *prototype design pattern*, from a Collection of prototype facilities held in the static `ProductPool` of the `Game` class. See details on `ProductPool` and `ProductPoolProduct` for more information. For this reason, this class implements `Cloneable`.

Farm, Mine and Refinery These are sub-classes of `Facility`, and over-ride `produce()` to increment a `Planet`'s resources when invoked. A `Farm` can be 'built' on a `Planet` to produce *food*, a `Mine` to produce *iron* and a `Refinery` to produce fuel.

Factory This class 'manufactures' `Ship`s. When created, the class is provided with a `productPoolIndex` which refers to an index into a static array of `ProductPool`s which are defined and initialised in the class. Using the *prototype design pattern* in a similar way to `Facility` construction, `Ship`s can be produced by invoking the `manufacture(int productPoolProductIndex)` method that places the product index into a manufacture queue. Within the `produce()` method, the index at the head of the manufacture queue is used to select a `ProductPoolProduct` for production, and checks how many 'ticks' should pass before the item should be *cloned* and added to the user's resources. When this number of 'ticks' has passed, and if the user has enough resources, the payment is taken, and the `Ship` is cloned, and added to a new `Convoy` at the location of the `Planet`.

ProductPool This class contains a Collection of `ProductPoolProduct`s. The class exists to act as a 'pool' of prototypes for replication by some *producer*. This may be either a `Factory` or a `Game`, producing `Ship`s and `Factory`s respectively. This implementation allows for several instances of *producers* to share the same `ProductPool`, allowing the possible produce of a producer to be defined very quickly and efficiently.

ProductPoolProduct This class holds cost and replication information for a *product*. Specifically, it holds the gold, food, fuel and iron costs to produce the `product` attribute. The `product` attribute must implement `Cloneable` and serves as a prototype for objects-to-be. The class also holds as an attribute how long the product will take to produce.

4.2 The `lispparse` Package

This package implements the Lisp parsing and unparsing routines for the communication protocol used by the project. It also gives the data types used to store Lisp expressions. It additionally provides a number of convenience methods which are use for constructing and retrieving information from Lisp expressions. It is used by both the client and server in order to construct and exchange messages. Some of the key classes in this package include:

Form (and its subclasses) The abstract data type for implementing Lisp expressions. A Form can unparse itself (i.e. convert into a string) as well as provides data about its attributes. For example a *ConsForm* consists of a `car` and `cdr`, which can be retrieved with `getCar()` and `getCdr()`.

Parser The main class which implements Lisp expression parsing. The main method `parseForm()` will read an expression from the input stream it was given as a constructor and return the parsed expression. It is a simple LL(1) parser which uses the `StreamTokenizer` class as its lexical analyser.

LispUtils This is a class of static utility methods which are used throughout the project. These are reimplementations of important Lisp utility methods, used by lispers to manipulate Lisp expressions, including `nth`, which returns the *n*th item in a given list, and `getf` which retrieves the corresponding value to a key in a Lisp-style associative array.

4.3 The `net` Package

This package implements the server and client pair applications for the game. As illustrated in Figure 1, each `Client` is associated with a `Service` which is provided by the `Server`.

GameApplication An abstract class for substitution of the *GameServer* and the *GameClient* classes.

GameServer It is a subclass of *GameApplication*. Basically, its jobs include maintaining a *Game* and a number of `Service`s. When a server starts, it will either create a new game or load a previously saved game from the database. When the server stops, it will save the whole game to the database for future reference. Each time when a client establishes a connection to the server, a service will be created and added to the services list; and when a client closes the connection, the corresponding service will be closed and removed from the services list.

Service This class is responsible for establishing and maintaining a channel for the communications between its corresponding client and the server. It receives messages sent from the client and forwards them to the server (which will then pass the messages and the service to the `ServerCommandInterpreter`). It also forwards the responding messages by the server to the client.

GameClient It is the other subclass of *GameApplication*. It works closely with the client side user interface. It sends the commands of manipulating the game as messages to the the server and receives notification messages from the server.

FormSocket This class encapsulates a *Socket* and its `Input/Output Stream` for sending and receiving messages in the format of *Form*s.

ServerException and ClientException Representing the exceptions which may be thrown by the *GameServer* and *GameClient*.

4.4 The `util` Package

This package includes a few classes which hold the global constants and variables which can be used by classes in all other packages, and define a number of global methods and functions for manipulating **Hibernate** `Session`s and drawing 3D graphics. It also holds the icons and images that are used in the game graphic user interfaces.

ConstantsAndVariables It holds the constants and variables (e.g. server address, port, etc.) that are needed for configuring the game and game applications.

HibernateUtil This classes defines a set of methods that make life easier while using **Hibernate**, such as opening and closing the `Session`.

MatrixUtil This class is used for performing 3D graphic transformation.

IconsAndImages All globally used icons and images will be held in this class. This class follows the **Skeleton Design Pattern**.

4.5 The `command` Package

This package includes a number of classes used by both the client and server in order to deal with incoming commands. It does this by allowing the package user to specify an associative array (a `Map`) of commands which are invoked whenever a request arrives. The key is the `car` of the request expression, and the value is the command to execute. Since the associative array is arbitrary and is retrieved at every request, the client and server may implement their own algorithms for determining the correct set of valid requests at any time. The key classes in this package include:

Command The superclass of all commands.

CommandInterpreter Given a `FormSocket` implements the interpretation of commands as described above.

4.6 The `server.command` Package

This package implements all the `ServerCommand`s which are the subclasses of the `Command`. It also implements the `ServerCommandInterpreter`.

ServerCommand and Its Subclasses They are responsible for handling the corresponding client command messages received by the server.

ServerCommandInterpreter This class stores all the valid `ServerCommand`s into a `Hashtable`. When the server passes a client command message to this interpreter, it will first parse the message to get the command name, get the corresponding `ServerCommand` from the table, and then pass the *parameters* extracted from the message and the `FormSocket` associated with the *service* to the *command* class, and finally invoke the `execute()` method of the `Command`.

4.7 The `www` and `www.servlet` Packages

This packages implement the web server portion of the game. The classes are responsible for invoking Tomcat in embedded mode and implementing the servlets used to present the web-based interface to the user. The web interface uses the Model-View-Controller (MVC) pattern in order to cleanly separate the code which retrieves information from the database and the code which outputs the HTML code for the generated pages. In this instance the servlets are the Model and Controller: they generate display data, possibly based on form inputs, and pass it to a JSP [3] which acts as the View. The JSPs mostly consist of HTML and contain as little Java code as possible in order to implement the View. Key classes in this package include:

WebServer Implements the invocation of Tomcat in embedded mode, using the `Embedded` convenience class provided by Tomcat. It sets up a web application rooted at `webapps/trader`, which is the directory containing the JSPs and `web.xml` file.

TraderServlet the common superclass of all the other servlets. It provides a number of convenience methods and defines a method `setRequestAttributes` which should be overridden by subclasses in order to provide the necessary information to the JSP View.

4.8 The `client` Package

This class implements a general client-side applet. Its main function is the implementation of the concepts of screens and subscreens within a client frame. The key classes include:

Screen The common superclass of any screen. Any subclass must implement the method `fillCommandMap` which informs the incoming message handler of which `Command`s correspond to which message.

Subscreen The common superclass of any subscreen. It is a subclass of `Screen` and acts as an indicator that this screen may be used as a subscreen.

SubscreenHolder An interface which must be implemented by any `Screen` which supports subscreens. The method `getSubContainer` must be implemented, which specifies a `JComponent` into which subscreens are placed.

ScreenChangeable An interface that must be implemented by the component which manages `Screen`s and `Subscreen`s. It specifies the methods `pushScreen`, `popScreen`, `setSubscreen` etc., which are used for switching screens and subscreens.

4.9 The `client.applet` Package

This package implements the specific client-side applet which is used for the game. It implements each of the screens and subscreens which are visible in the applet. Each screen or subscreen handles its own relevant messages while it is visible via implementation of the `fillCommandMap` method. For example the main screen is implemented by `MainScreen`, a subclass of `Screen`, and the convoy selection subscreen is implemented by `ConvoySelectSubscreen` a subclass of `Subscreen`. This package also implements some of the custom views, subclasses of `Canvas`, which are used to provide a richer user interface. For example `PlanetView` implements the 3D view of the planets in a solar system which is used to select a planet.

5 Conclusion

After three weeks of intensive work, our group is pretty happy with what we have done so far – we have a fully working game now. Most of the functions (e.g. mailing service, broadcast service, building facilities, trading goods, travelling in the Universe and so on) of the game have been implemented. Both the server side and client side applications have been tested. A web interface for registering new players, logging in and joining the game has also been produced. The game is now available for external users to play and evaluate, and we are preparing for the next version of the game.

However, due to time limitations, **fighting** *between players' convoys* has not been implemented. But this function would be relatively easy to implement if we were given enough time. In a future version of the game, we will add more rules to it, and perhaps expand the Universe and allow players to produce more different types of goods and trade them.

Over the past three weeks, in order to meet the deadline, all of our group members have been working hard day and night. In return, we have learnt quite a lot of tools and techniques for producing a good project. In summary, here is a list of what we have learnt and/or used:

- **ArgoUML** for the game design.
- **Trac** and **Subversion** for group project development management.
- **IRC** for group communication.
- **Lisp** syntax for implementing the protocol of communication messages sent between the server and clients.
- The `java.net` package and **Socket Programming** for implementing the *Server* and *Client* pair.
- Parts of **J2EE** (*Java Servlets* and *JDBC*) for implementing the game and the web interface.
- Advanced **Java Swing** techniques for designing good user interfaces.
- **Hibernate** for writing persistent classes (see section 3.3).
- **Javadoc** for producing *API* documentation
- **LaTeX** for producing this report
- and last but not least, *communication* and *teamwork* skills.

References

[1] Hibernate. Internet, 2005. `http://www.hibernate.org/`.

[2] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Technical Report Internet RFC 2616, IETF, 1998. `http://www.ietf.org/rfc/rfc2616.txt`.

[3] SUN. Java server pages. Internet, 2003. `http://java.sun.com/products/jsp/`.