Anna Cubed

Ben Campbell Daniel Piddock Alex Pounds

20060301

Abstract

Solo is a card game similar to Whist and Bridge. The goals of the game are very variable and flexible, making it an interesting and flexible challenge to develop a playable version on computer. A server was created to run the game, as were numerous clients to play the game. Computer controlled opponents ("bots") were also created.

1 Introduction

Solo is a trick-taking card game for 4 players. It can be thought of as a little like Whist with bidding; a passing familiarity with the game is highly recommended before proceeding further (see the "Solo Rules" user handout).

One of the group members spent a lot of his time at college playing the game of Solo with his friends and had always thought it would make an interesting computer game. This appreciation for the complexity of the problem to be solved led to the suggestion of the project and its ultimate acceptance.

At the start of each hand players bid to say how many tricks they will win. Common bets include **Prop** ("With someone else, I will make 8 tricks") and **Cop** (Only available after someone has said **Prop**; it means "With you as a partner, we will make 8 tricks"). **Solo** ("I will make 5 tricks on my own"), **Misere** ("On my own with no trumps I will lose all the tricks"), and **Abundance** ("I will win 9 tricks on my own if I can pick trumps") are also common bets. After the bidding the game commences. If the winning bidder makes their contract points are awarded according to the difficulty of their bid.

This concise description illustrates some of the challenges inherent in implementing the game. Players play either on their own, in explicit partnerships or implicit groups. Adjudicating bids is non-trivial and computer-controlled players must be able to evaluate the strength of their hand and bid accordingly. Computer-controlled players should be able to play cards with some semblance of strategy. As the game features multiple players there are possible race conditions and the challenge of parallel programming. During the course of the project we encountered all of these issues and successfully overcame them.

2 General Architecture

The decision was made early on to implement our system in Java. This platform was chosen for its portability, strictness of syntax, and full API. Portability and openness was an important consideration, both due to the beliefs of the group and the differing environments of each member. As a result we ensured that other parts of the system could interact with non-Java programs; a plaintext network protocol was devised for this purpose.

Java 5 is required for Anna Cubed to compile and run. The implementation makes extensive use of the new features in the latest revision to the Java language – indeed, some of its limitations were revealed during development. Packaging was used to keep related areas contained and enable sharing of classes, interfaces, and enumerations where necessary.

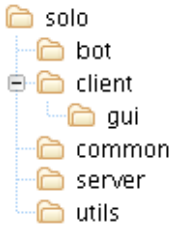

Figure 1: Package structure of Anna Cubed.

The system works on a client-server architecture. A game server is required to play a game; it is tasked with keeping scores, adjudicating bidding and play, and managing users. Clients connect to the server as either a player or a bot.

To enable this several interfaces were created (see the package `solo.common`), including `solo.common.Client`, `solo.common.Bot` and so on. A communications layer was also designed to ease the later introduction of network play but the eventual implementation took a different direction (see section 6, *Deviations from Original Design*).

When a player starts a client they have the option of connecting to an existing server via the network or starting a server themselves. For a local game, the server fills the remaining places at the table with bots. In a network game the master player (the player who started the server) may click "begin" at any time; any unfilled seats at the table are taken by bots. The server has responsibility for prompting the clients when a response is needed and transmitting this response (suitably sanitised) to other players. The server will also inform the client if it makes an invalid play/bid, allowing it the option to recover from this.

3 Development Process

One of our key concerns from the start was the best way to develop the project. Our group was small enough that many project management methodologies would be overkill, but it was not small enough and close-knit enough to be left unmanaged. As a result we decided on semi-fluid job roles; we would all work on the implementation, but in addition one member acted as *project manager*, one as *lead architect*, and one as *quality assurance*. As it turned out the project leant itself nicely to three developers; one member took responsibility for the GUI client, one for the server, and one for the bots.

For the first few weeks of the project the group had many meetings to discuss the project's scope and design the basic architecture. This was broken down into packages that were further broken down into classes. The lead architect drew up some specifications and UML-like diagrams; once these had been approved Java interfaces were authored for the implementations (see the interfaces in `solo.common`). After the first few weeks the design decisions had settled down and the weekly supervisor meetings were used as the main contact between the team. They were used to review progress and set goals for the next week. Instant messaging and email was used throughout the week to keep the team in contact. This was coordinated through the project manager who set the direction of development.

To protect against mistakes a CVS repository was used to store the project. As well as containing source code it also contained documentation and other support files. This had many benefits, such as being able to test in isolation and thus commit solid revisions to the "authoritative" version of the project and being able to roll back to earlier versions in the case that something broken slipped through. The project manager configured the repository to email the group whenever a CVS transaction occurred; this proved very useful as it kept the entire group informed about individual's development efforts.

The project build process was automated using `ant`. This allowed the project to be rebuilt from a single command as well as supporting targets to build jar files and generate javadoc documentation. `ant` is very good at ensuring the entire project is up to date, which makes tracking down bugs easier.

For quality assurance a number of different techniques were used. Test rigs were built to interrogate some of the basic classes; despite initial skepticism about their necessity they revealed several bugs that could have been uncaught otherwise. A similar test rig was built for the GUI; although this showed up fewer bugs it was still valuable. A different approach was used with the bot due to the details of its implementation.

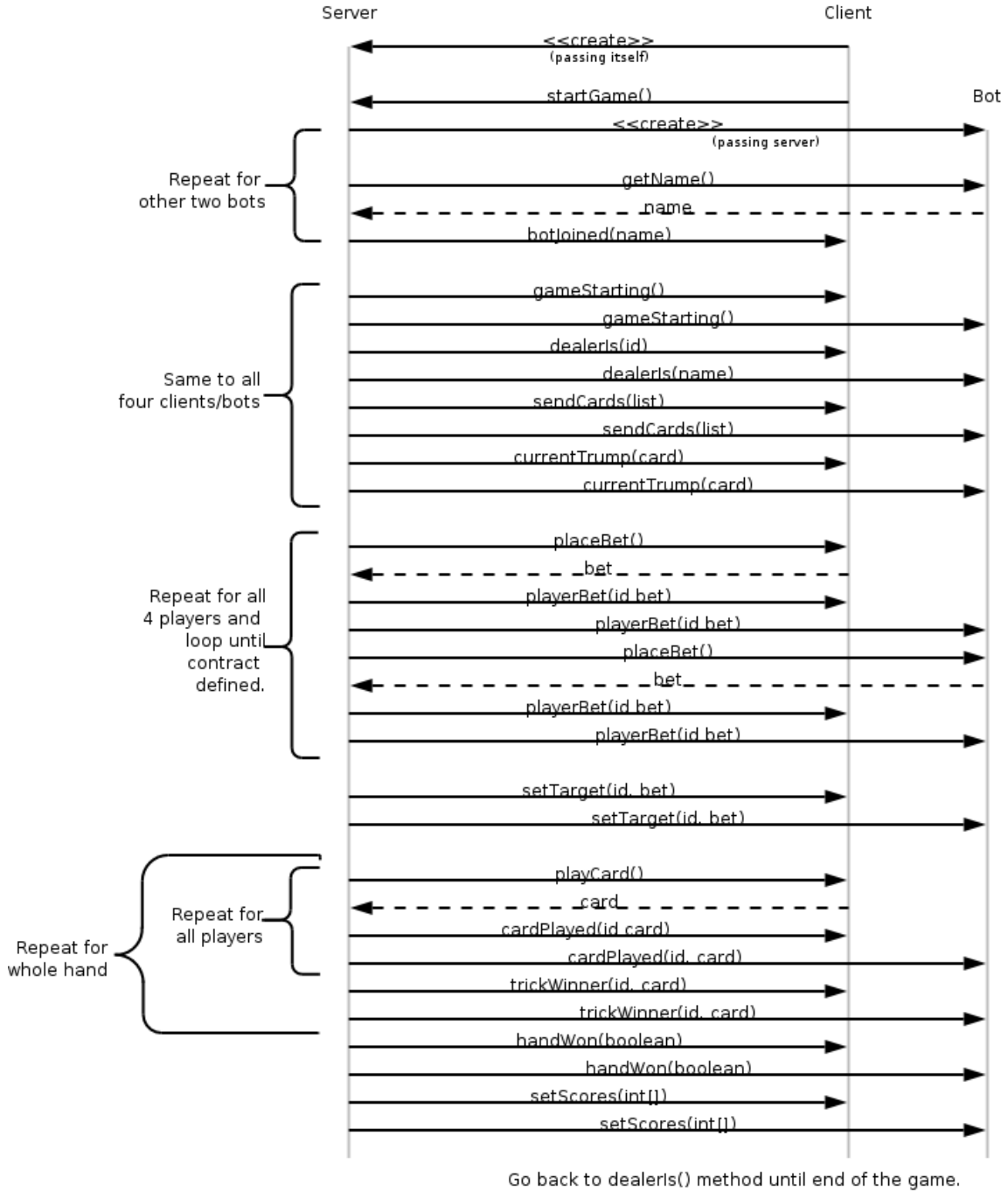

Figure 2: UML diagram of the messages passed between the client, server, and bots.

It interacts with the server using a simple, well-documented interface and most of its functionality is within private methods. It is also not entirely deterministic making automated testing problematic. As a result a separate developer who had nothing to do with its development audited it upon completion, examining it for any problematic areas and contributing fixes where necessary.

Towards the end of development the project was packaged and tried on many different platforms. After fixing a small issue caused by the threading model of Windows no more problems were encountered. Anna Cubed works as designed on Windows, Linux, Solaris, and Mac OSX.

4 Specifics

Below are specific details of our implementation. One problem with the game itself is that card games have an evolutionary history based on word-of-mouth propagation and the recollection of people. Unless a game is very popular (such as Bridge or Poker) there are no official boards setting rules. As a result *our* Solo might not be the same as *your* Solo. The rules in the "Solo Rules" handout should be considered canonical; they are based primarily on a book of card games[1] and a web-based source[2].

For details of the implementation, we urge you to look at the javadoc available on the CD. This includes details of the interfaces and how they should be employed as well as details of the implementation.

4.1 The Server

The server is a single-threaded application. Once the client has told the server to start the game there is no more external input (apart from the client telling the server to quit). The game is controlled by the single server class which calls methods on clients implementing the `Player` interface. These calls inform them of the game state or ask to take an action (such as playing a card or placing a bid).

The single-threaded design was chosen to avoid headaches in synchronising various threads and objects. Solo is a turn-based game so gameplay is unaffected by this design choice. By simplifying the implementation in this fashion we have created a more robust game server. The introduction of network play does mean that the server could hang waiting for input from a remote client but the group decided that this drawback did not outweigh the benefits.

The server stores details relevant to the game state as global variables inside itself. Any details specific to a player are stored in individual `PlayerContainer` objects. This keeps details grouped together and in easily accessed locations.

One area of note is that of getting the server to quit reliably. The serial and non-concurrent nature of the server meant that clients could not easily instruct the server to quit at any point. The solution was for the client to set a flag in the server that was checked before calling any methods on the clients. If the flag is set the server throws a `GameOverException`, a subclass of `RuntimeException`. Using the `RuntimeException` avoids the need to explicitly list the exception in all the method headers as it is always caught internally. The server informs the other players that someone has quit and then terminates itself.

Networking was implemented at a late stage but had been planned for from the start. To minimise modifications to the server a separate `NetworkServer` class was created to listen for connections and create `SocketPlayer`s. If a `Server` object is instantiated with a positive port number it creates a `NetworkServer` automatically. The server can treat a `SocketPlayer` just like any other player as it is a subclass of `Player` that handles all the network communication between the server and the remote player. The only modification to the `Server` class was the addition of a small method to register the `SocketPlayer`, assign it an ID number, and inform the player of other players.

The server stuck fairly closely to the original design, although it has evolved where necessary to work around unforeseen implementation issues. The creation of the server by the client and the client registering itself with the server is a good example of this.

4.2 The Client

Several clients were created during the course of the project. The first was a text-based client that presented users with a description of the game and allowed them to enter the card they wished

to play via the keyboard. It can be found in `solo.client.ClientTextUI`. Although it is most useful for debugging it is fully functional and supports network play.

Two graphical interfaces were produced; a prototype (Figure 3) that was discarded...

> "Plan to throw one away; you will anyway." [3]

... and a finished version that was built on its foundations. Both of these used Java SWING.

The `App` object holds the main method for the GUI. It creates a `ClientComm` object (used for communicating with the server and passing requests to the GUI) and a `GUI` object (which extends the `JFrame` class and takes a `Menu` object that creates the menus and controls their functionality).

The GUI initially contains a title panel with the title and authors of the game. When a game is started the content pane is set to a `ClientPanel` that holds the differing display elements for the game and passes variables from the main GUI to the individual components. Some of the key features of the `ClientPanel` (see also Figure 4) are:

- The `GameArea` object, holding textual information about the current round and containing the `PlayerDetails` objects. These hold information about individual players (such as their tricks, scores and current contract).
- The `Hand` objects, one for each player. The `Hand` class draws the hand of the player and set `ActionEvents` on each card that cause it to be passed to the GUI when clicked. After some experimentation with spreading hands out to fill the available space, hands were stacked to reduce the space required and keep them within a player's immediate focus. Hands of players may be revealed to others if a round is played under a misere ouverte contract.
- The log, a small text area that gives some feedback to the user about the game. This is not essential to playing the game but does provide some feedback about events not requiring a user's immediate attention. It would also allow a user to scroll up and see what cards have been played previously if they wish, while not making this feature overbearing. If the user was a purist they are likely to rely on their own recollection and consider the log cheating.

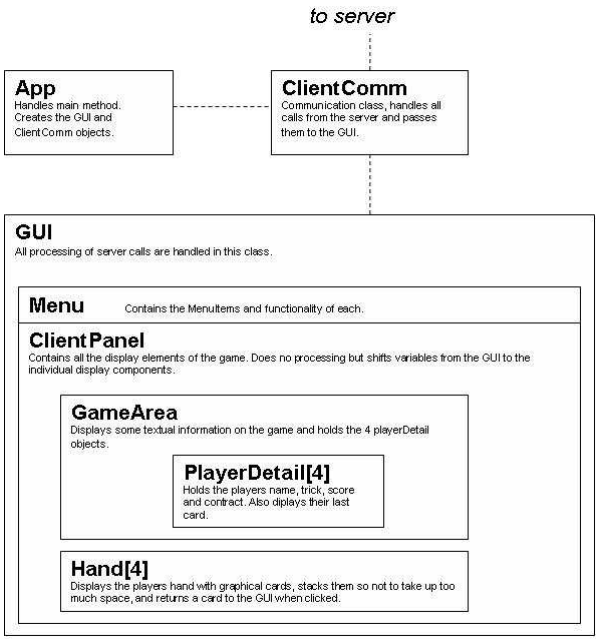

Figure 4: A high-level overview of the class relations in the GUI.

If the user chooses the "Play Network" option from the File menu the GUI's content pane is replaced with a `NetworkPanel` object. This allows the player to choose whether to start a server or join an existing game. If the user wishes to join the existing game the player may input details such as their name, the host and the port to which they wish to connect. Once the game has been created or connected to this `NetworkPanel` is disposed of and replaced with the `ClientPanel` described above.

4.3 The Bot

In some games (such as Blackjack) it is possible to "run the numbers" and generate an optimal strategy.[4][5] Unfortunately the search space for Solo is much, much larger. Generating an optimal strategy by crunching numbers would be

Figure 3: The GUI prototype.

an exercise in futility or at least a PhD-level problem.

As a result the advanced bot makes its card playing decisions according to what it can deduce about the other players' hands. It will detect when a player is definitely desuited in a given suit and use this information when deciding what to play or lead. It also uses information from its own hand and the cards it has seen to estimate how many cards other players hold in each suit and use these figures to make decisions too. If it is likely or possible to win the trick it will make a random decision whether to try and win the trick or not; from 90% of the time it will try to win if it is a fairly safe attempt to 33% if it is a risky move. The bot is also smart enough to try and desuit itself in its short suits, enabling it to trump earlier. When playing a card, it accounts for how likely it is to be trumped by a future player. It also understands the concept of **Prop** and **Cop** and will not win a trick that is already likely to be won by their partner.

The advanced bot is particularly good at playing for and against **Misere** and **Misere Ouverte**. These bids do not come up very often, but when they do the bot tends to perform extremely well (although there is a little room for improvement.) The server architect was staggered when, during testing, the bot bid **Misere**. He was floored when (despite his best efforts) it won the contract.

The bots run their hands through a scoring algorithm and use this score to bid. They will bid conservatively at first and more bullishly in later rounds. The scoring works fairly well, but could be tweaked to give more weight to longer suits and other factors that result in a more playable hand.

The practical upshot is that the advanced bot plays quite aggressively and will frequently win contracts. If the bot is allowed to lead frequently it will capitalise on this so a human's best tactic is to play aggressively and remain in control as much as possible. Of course, strong play is also a viable option; a strong hand puts a player at a definite advantage. The simple bot follows a very basic strategy – follow suit if it has to, trump if it can, play a random card otherwise – and is included for comparison purposes.

4.4 Utilities

One utility program was produced, found in `solo.utils`. It constructs random hands and runs them through the bot's scoring algorithm, generating statistics on the hand strengths as it goes. It was used to aid with picking scores for the bid boundaries of the bot; while it is fairly straightforward to design criteria for positive and negative aspects of hands, picturing how this will effect the end score of a hand is a challenge. `solo.utils.HandWeights` helped pick the bid boundaries. Had we been able to implement non-traditional bots in the time given, this class could have formed part of a genetic algorithm or neural networks implementation[6]. Scoring of hands could have been altered and processed, then evaluated for accuracy.

4.5 Network Protocol

The network protocol borrows from SMTP, in that it uses 4 letter command words followed by parameters. This provides for some human readability as well as easy parsing. Details of the protocol can be found in the separate network protocol document. Discussion of the networking itself can be found above and below in section 6 (*Deviations from Original Design*).

5 Playing the Game

If necessary, the game must be compiled first. This requires Java 5 and `ant`. `ant` takes care of compiling all the sources and creates a jar file in `../dist`. This can be run with the command `java -jar AnnaCubed.jar` when inside the dist directory. Running the jar in this manner starts the GUI; if a user wishes to interact with the classes instead of the jar they can find this GUI at `solo.client.gui.App`.

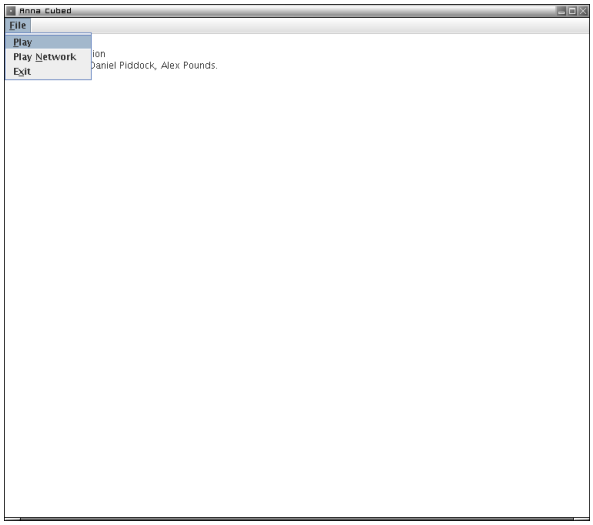

Figure 5: The initial screen, with the File menu visible.

On starting a game a basic interface is shown. The user should select either "Play Local Game" or "Play Network Game" from the file menu. If a local game is chosen the game will begin immediately; otherwise the user has the option of hosting a game or joining an existing game.

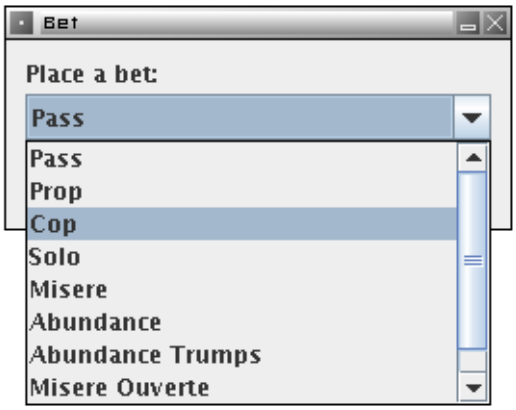

Figure 6: Placing a bet dialog.

Once the game has begun the user is prompted to enter a bet. They should evaluate their own hand and consider its strength, along with the bets

entered by the bots. Once the betting round is over the play begins.

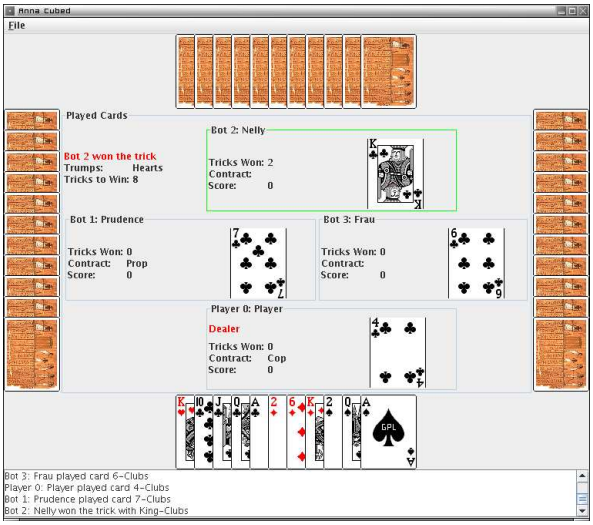

Figure 7: A game in progress; Bot 2 has just won this trick.

The player highlighted in red is the current player and needs to play a card by clicking on it in his hand. If the player plays an invalid card he is informed of this via a dialog box; otherwise the card is played. The trick winner is indicated momentarily by highlighting the winning player in green and their trick count is incremented. The current contract is noted in the top left, as well as in the players' information boxes.

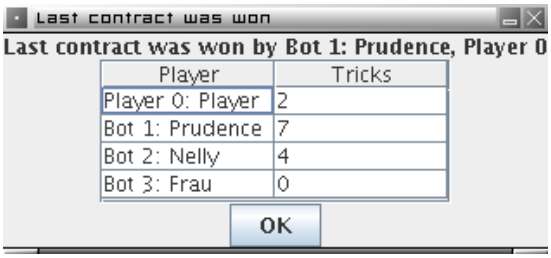

Figure 8: The hand scores at the end of a round. The players made their contract.

When the hand is over the counts of tricks made are displayed in a dialog box along with an indication of whether the current contract was won or lost. Points are awarded accordingly and the next hand begins at the bidding stage.

6 Deviations from Original Design

Reviewing the original design documents it can be seen that most of the design goals were met and the overall design did not move far from the original. Despite this some parts of the design had to be reworked once implementation began. The best example for this is the removal of the `CommModule` class that was going to sit between the `Server` and `Clients`. The plan was for the `CommModule` to provide a transparent interface between the server and clients and add networking to it at a later date. Instead local games were prioritised over network play which made `CommModule` a redundant class that would simple relay method calls.

Rather than have this intermediate class the design was altered so that after `Client` had created the `Server` the server would make calls on `Player` subclasses. Thus a client never has to call any methods – it merely takes input and returns data where necessary. This had a side benefit of making implementing networking very straightforward: the server would still talk to a `Player` object, `SocketPlayer`, as normal but that player would be sending the data over a TCP/IP link to a remote client. The client returns data to the `Player` which passes it back to the server.

Adding networking to the clients was very similar. Both the text and GUI clients use the same `solo.client.NetworkClient` class. Small modifications to both clients were needed so they could join and host network games but once the `NetworkClient` has been instantiated there is no difference to running the server locally.

The original design included the ability to have spectators – clients that watch the game and have no input – but these were never implemented. They could be added at a later date; the `solo.common.Client` interface only receives data relevant to all players. Player implementations must implement the `solo.common.Player` interface (which extends

`solo.common.Client`). Whilst the design supports the addition of observers the server implementation would need some modification before they could be used.

The original interface plans have had minimal modifications. Player ID numbers were used to identify clients instead of `String` names as they were easy to compare and easier to keep unique. Player names are still supported but the implementation uses ID numbers to identify players throughout. Another change was the removal of the `conflictInBet` method in `Client` in favour of informing the client that they had placed an invalid bet or inviting a client to upgrade their bet using the `upgradeBet` method.

7 Known Issues

- Scoring is not entirely in line with the published rule sources[2][1]. This is not a major issue - the scoring is accurate, just not necessarily the same as other implementations. The server handles the scoring and pushes it to clients so players will not have different concepts of the score.

- It would be possible to implement a bot that works more on published strategy rather than its own estimations of what makes sense. This would make it stronger but also an order of magnitude more complex. The bot provides an interesting game as it is and is certainly no pushover, so this is more of a possible enhancement than an issue. It is also likely to be stymied by players that do not play with any higher strategic thought.

- The GUI could do with a little more user feedback; making it clear about how network joins are proceeding would be nice (although connection errors are displayed), as would an indication of who is currently winning the bidding.

- When the user is prompted to enter their bid the dialog covers the bids of one of the players. Unfortunately this is down to Java SWING and is outside of our control. This dialog will also show bids that are unavailable; these should either be removed, disabled, or marked in some way.

8 Conclusion

Through good project management and a solid initial design we have implemented a quality card game for one player or multiple players via a network. Although we have deviated slightly from the original plans this has served to produce a more tightly-focused final year project. As well as highlighting our development skills it also shows our ability to work as a team whilst maintaining responsibility for disparate areas of the project. The departure of our original supervisor also makes the project a testament to our adaptability.

Anna Cubed is in one sense "just a game". In another it is so much more. We have created a server, several clients, two bots, and an extensible framework/design to allow others to work with the system. The well-defined and well-documented interfaces make creating other clients and players straightforward.

Games at this level are very rarely groundbreaking but Solo is a fairly obscure field. The group is aware of only one shareware Windows implementation that is unmaintained – it has no website and has not been updated since 1995. Anna Cubed is the only current implementation of Solo and is unique in its field: a cross-platform multiplayer Solo game.

References

[1] Jacqueline Harrod Arnold Marks. *Card Games Made Easy.* Clarion, 1997.

[2] David Morrison et al. Frank O'Shaughnessy. *Pagat.com: Rules of Solo Whist.*

[3] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Professional, 1975.

[4] Martin H Millman. A statistical analysis of casino blackjack. *American Mathematical Monthly*, 90:431–436, 1983.

[5] Roger Williams. *A Casino Odyssey, Part Two.* 2001.

[6] Andrés Pérez-Uribe and Eduardo Sanchez. Blackjack as a test bed for learning strategies in neural networks. *Proc. IEEE Intl. Joint Conf. on Neural Networks IJCNN'98, Anchorage*, 3:2022–2027, 1998.