Deuce

The Java Big Two Game

by Xida Chen, Jeffrey Lo, and Felix Wong

Crux Research Facility

`http://cx.freeshell.org/big2/`

Printed in Canada

10 9 8 7 6 5 4 3 2 1

To those who spent days and nights testing this game.

Acknowledgments

This project would not be possible without the enthusiasm and help of Sammy Leong, who was invaluable in ironing out the technical details of Java, testing the game, and providing suggestions on the design of the game.

I am also in debt to Mr. Trudeau, who provided instructions and help throughout the development of this project and provided me with a Mac-OS-X-based lab facility.

Finally, I would like to thank the following generous people for testing this game:

Andy Lau, Jeffrey Lo, Felix Wong

Preface

Big Two is a simple and entertaining card game. I shared many of my laughs with my friends when we played this game. We have always enjoyed it. Unfortunately, I really wanted to play this game online. Despite my effort to search for this game, I was unable to find it. Perhaps it didn't exist online. I then decided to make this game myself. I thought that it wouldn't be too hard.

I started this project using Java, and not long I realized that I was half correct. The game structure itself wasn't really difficult, but the making of the game was much harder than I anticipated. The difficulty really started to arise when I began to program the client-server part of the game. It was like adding a whole new dimension to the entire game structure. When clients are interacting with the server, so many things could go on all at the same time. At times, I realized that I really had to think in three-dimensional terms in order to get the whole picture about what was going on, with both the client side and the server side.

To me, the greater difficulty usually means I have to spend more time on it, and I did. I spent most of my Winter break researching about the obstacles I was facing at that time; I overcame one after another. I tried to make this game fast and easy for the user, but sophisticated inside.

Here I am pleased to present you with the final product. This game and its supporting Web site were coded entirely in Vi, an UNIX plain text editor. The game was compiled using Java 2 SDK version 1.4.1; the version control was done by the Concurrent Version System (CVS). The Web pages were written in XHTML 1.0 strict, and this user manual was written in $\LaTeX\,2_\varepsilon$.

Again, I hope that you enjoy this game as much as I do, and I hope you find this manual helpful. If you have any suggestions, please email me at `cx@maclab.org`.

X. C., Toronto, 2003

Contents

Chapter 1

The Rules

1.1 History of Big Two

The card game Big Two probably originated in coastal China; in the late twentieth century it became very popular in Shanghai, Hong Kong, Taiwan and also in the Philippines and Singapore; it also spread to some western countries.

1.2 Players and Cards

The game is best for four players, each playing for themselves.

A standard 52 card pack is used, the cards ranking from high to low: 2-A-K-Q-J-10-9-8-7-6-5-4-3. There is also an ordering of suits – from high to low: spades, hearts, clubs, diamonds.

1.3 Object of the Game

The object of the game is to be the first to get rid of all of your cards, by playing them to the table. Cards can be played singly or in certain combinations. If you cannot be the first to play all your cards, then your aim is to have as few cards as possible when another player finishes.

1.4 Playable Combinations

There are four types of legal play: single cards, pairs, triples and five-card groups.

1.4.1 Single cards

These rank from two (high) down to three (low), and between cards of the same rank a higher suit beats a lower suit.

1.4.2 Pairs

If there is a pair of equally-ranked cards, then twos are the highest and threes are the lowest. Any higher ranked pair beats one with a lower rank. Between equally-ranked pairs, the one containing the highest suit is better – for example ♠9-♢9 beats ♡9-♣9 because the spade is higher than the heart, but ♣Q-♢Q beats ♠J-♡J because queens beat jacks.

1.4.3 Triples

If there are three equally-ranked cards, then three twos are the highest, then aces, kings, etc. down to three threes, which is the lowest triple.

1.4.4 Five card groups

There are five types of playable five-card combinations. In ascending order, they are: straight, flush, full house, four of a kind, straight flush.

1. A *Straight* consists of five cards of consecutive rank with mixed suits. A straight with higher ranks beats a lower one, irrespective of the suits of the cards. When the ranks are the same, the suit of the top card determines which is higher. So for example, ♢K-♣Q-♣J-♡10-♡9 beats ♠Q-♢J-♢10-♢9-♢8, which beats ♢Q-♠J-♠10-♠9-♠8. Twos do not rank high in straights, but rank below the 3, so the highest straight is A-K-Q-J-10 including the ace of spades. Aces can be counted as low to make the lowest type of straight 5-4-3-2-A, which is beaten by 6-5-4-3-2 and 7-6-5-4-3. An ace can be used at either end of a straight, but not in the middle, so collections like 3-2-A-K-Q or 2-A-K-Q-J are not valid straights.

2. A *Flush* consists of any five cards of the same suit. A flush in a higher suit beats a flush in a lower suit, irrespective of the ranks of the cards. Between two flushes in the same suit, the one with the higher top card is better. So for example, ♡9-♡7-♡6-♡5-♡3 beats ♣2-♣J-♣9-♣6-♣4, which beats ♣A-♣K-♣Q-♣10-♣7.

3. A *Full House* consists of three cards of one rank and two of another rank; between two full houses, the one whose triple is of higher rank is better. So for example, 9-9-9-4-4 beats 8-8-8-K-K.

4. *Four of a kind* (or *quads*) consists of all four cards of one rank, plus any fifth card. The fifth card must be included – four equal cards by themselves are not a playable combination. Between two fours of a kind, the rank of the four cards determines which is higher.

5. A *Straight Flush* consists of five consecutive cards of the same suit, twos ranking below threes and aces ranking high or low, as in straights. The rank of the highest card determines which of two straight flushes is higher; between two equal ranked straight flushes, the one in the higher suit is better, so the Royal Flush in spades ♠A-♠K-♠Q-♠J-♠10 is the highest straight flush and the best five-card combination.

A combination can only be beaten by a better combination with the *same* number of cards, so a single card can only be beaten by a single card, a pair by a better pair and a triple by a better triple. You cannot, for example, use a triple to beat a pair or use a straight to beat a triple. However, a five-card group can be beaten by a five-card group of a stronger type – any flush beats any straight, any full house beats any straight or flush, any four of a kind plus an odd card beats any straight, a flush or full house and any straight flush beats all of the other types of five-card groups.

Note that although the playable combinations are similar to poker hands, there are important differences. For example, there is no “two pair” combination, and although a four of a kind requires a fifth card to complete the combination, a triple cannot be accompanied by extra cards (unless of course these make it into quads or a full house).

1.5 The Deal

Dealing and playing are normally done in an anti-clockwise way. Any player may deal first; thereafter the winner of each hand deals the next. The dealer shuffles and the player to the dealer’s right cuts.

The dealer deals out the cards, one at a time, starting with the player to the dealer’s right and continuing counterclockwise until all the cards are dealt. Everyone will have a hand of 13 cards, which they can look at and sort.

1.6 The Play

In the first deal of a session, the player who holds the three of diamonds begins and must play this card, either by itself or as part of a legal combination. Thereafter the winner of each hand plays first in the next rounds. The person to this player's right plays next, and so on round the table. At your turn you must either pass (play no cards) or beat the previous play by playing a higher combination of the same number of cards. All plays are made by placing the cards face up in the center of the table, so that a heap is formed. This continues for several circuits if necessary, until all but one of the players pass in succession, with no one being able or willing to beat the last play. When this happens, the heap of played cards is set aside face down (or in some places, the players just leave all the played cards in a face-up heap on the table). The person who played highest (all the others having passed) begins again by playing any card or legal combination of cards.

You are never under any obligation to beat a card or set of cards just because you are able to – you may always choose to pass and keep your high cards for a better opportunity. Passing does not prevent you from playing when your turn comes round again. *Example*: At a late stage in a game, South starts with a 4, East beats it with a jack, you (North) pass, West plays an ace, South and East pass. You suspect that West will be able to win by playing all of his/her remaining cards as a group if you pass, so you now play your ♣2, which you held back before.

Everyone is allowed to know how many cards the other players have in their hands at any time – if asked, you must answer truthfully.

The first player who succeeds in playing all the cards in his/her hand wins. As soon as this happens, the play ends and the hand is scored.

1.7 Scoring

The players other than the winner score penalty points for the cards remaining in their hands. The penalty is one point per card in your hand if you have 9 cards or fewer, two points per card if you have 10, 11 or 12 cards, and three points per card (i.e. 39 points) if you have all 13 of your cards left at the end, because you never played any cards at all. The winner, having no cards, gets no penalty points for the hand.

This game is often played for money. In this case, before starting to play, the players agree on a stake – for example $1 per point – and at the end of the session each pair of players settles up according to the difference between their scores.

Chapter 2

Game Instructions

2.1 System Requirements

Deuce is a multi-platform Java game that runs in your existing browsers. You do not have to download the game to your local disk and install it. However, you do need the *Java Plug-in* (*Java Run-time Environment* from Sun Microsystem) for your browser in order to play the game. It is installed in many systems by default. If you do not have it in your system, or if you have an old or incompatible version, you can download the latest version from `http://java.sun.com/getjava/`. You will need at least an 800×600 screen resolution to display the entire game on your screen. Please read the FAQ page in the Deuce Web site for frequently asked questions about the game. Its link is provided in the next section.

The game was tested successfully on the following systems:

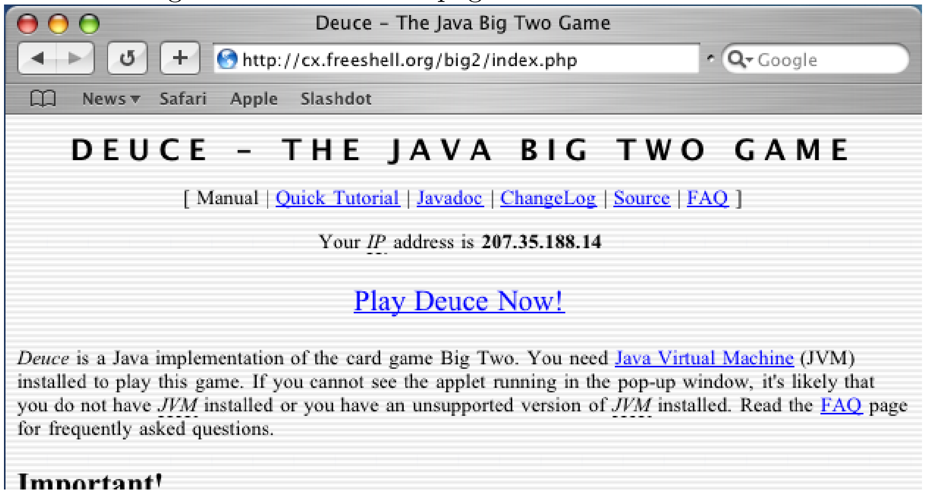
- a Mac OS X box running Internet Explorer 5.2 and Mozilla 1.2 with JRE 1.3;
- a Linux box running Mozilla 1.0 with JRE 1.4;
- a Windows XP box running Internet Explorer 6 with JRE 1.4.

2.2 Entering the Deuce Page

To play the game, visit `http://cx.freeshell.org/big2/` in a Java-enabled browser.

After seeing the Flash intro of the game, you will be taken to the main page of the Web site (Fig. 2.1) where you can load the game simply by clicking on “Play Deuce Now!”. A new browser window will pop up.

Figure 2.1: The Deuce page on the World Wide Web.


Deuce - The Java Big Two Game
http://cx.freeshell.org/big2/index.php
Google
News Safari Apple Slashdot

DEUCE - THE JAVA BIG TWO GAME

[Manual | Quick Tutorial | Javadoc | ChangeLog | Source | FAQ]

Your IP address is **207.35.188.14**

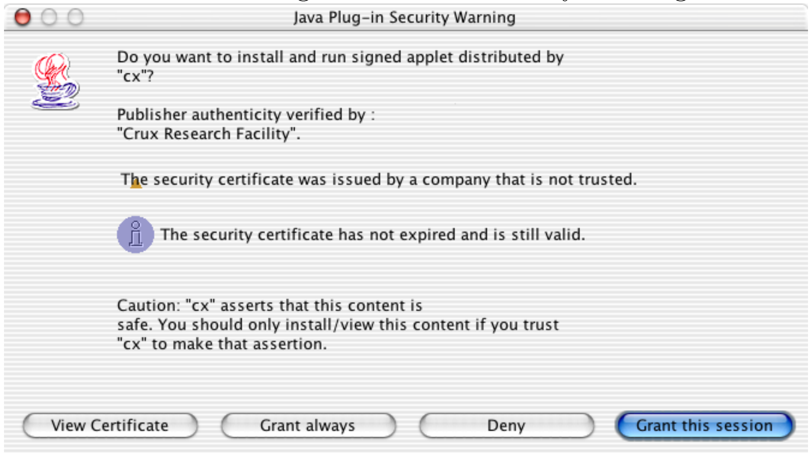
Play Deuce Now!

Deuce is a Java implementation of the card game Big Two. You need Java Virtual Machine (JVM) installed to play this game. If you cannot see the applet running in the pop-up window, it's likely that you do not have JVM installed or you have an unsupported version of JVM installed. Read the FAQ page for frequently asked questions.

Important!

2.3 Loading

Figure 2.2: The security warning.



Do you want to install and run signed applet distributed by "cx"?

Publisher authenticity verified by : "Crux Research Facility".

The security certificate was issued by a company that is not trusted.

The security certificate has not expired and is still valid.

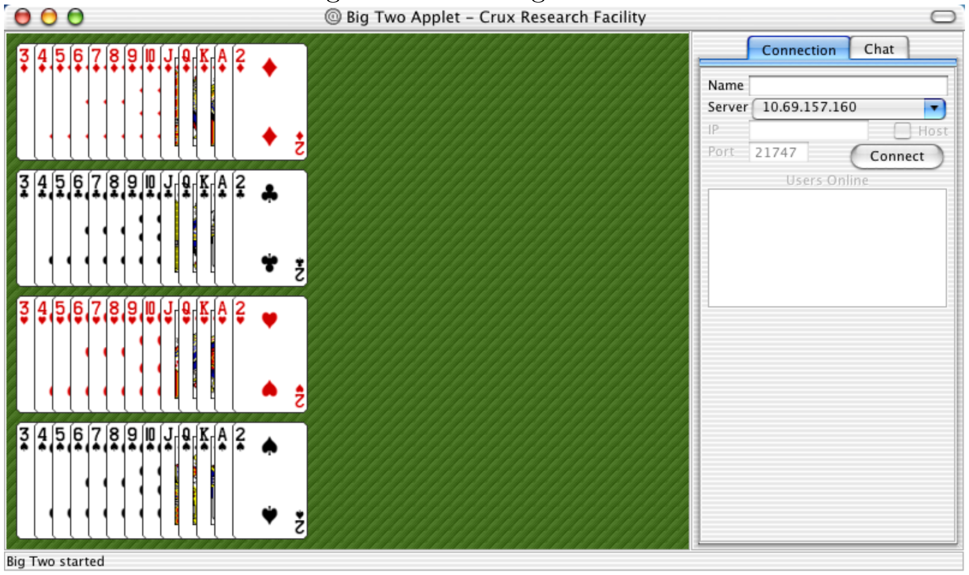
Caution: "cx" asserts that this content is safe. You should only install/view this content if you trust "cx" to make that assertion.

After the game finishes loading, a security warning will pop up (Fig. 2.2). It will ask you if you wish to install and run the signed applet. You will then have to click on "Grant this session" or something equivalent (depending on the version of Java you have) in order to have networking enabled in the game[1]. After you click on that, the game will then be loaded. (Fig. 2.3).

[1]The digital signature is required because you will have to connect to a foreign Deuce

You will see the intro screen with all fifty-two cards. This is when you

Figure 2.3: The login screen.



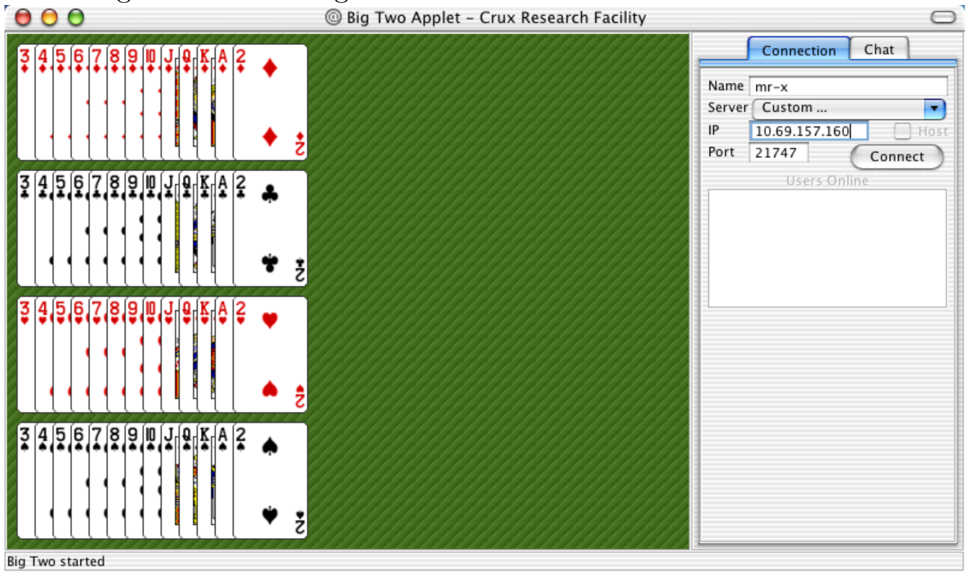
connect to the server.

2.4 Logging In

To connect, you have to enter the name you wish to log in as. A legal name can contain two to ten characters (alphanumerics, dashes, and/or underscores). Below the name field, there is a pull-down menu where you can select the standard server you wish to log into, or you can select the option "Custom..." to connect to a server that is not listed (For example, your friend is hosting a Deuce server and you want to connect to that computer. Hosting is described in Section 2.8). To connect to a custom server, enter its IP address after selecting "Custom..." from the pull-down menu; see Fig. 2.4. (Domain name is also accepted if the server has one.) The default port should be used in most cases unless the server is running under another port, which in this case, you enter the port that the server is running on. After all of that is completed, click on the "Connect" button and you will

server other than the Web server where the game is located. The Java limits the ability of an unsigned applet for security concerns, mainly because it can cause malicious code to be loaded from a browser.

Figure 2.4: Entering the IP address of a remote Deuce server.


Big Two Applet – Crux Research Facility

Name: mr-x
Server: Custom ...
IP: 10.69.157.160
Port: 21747

Big Two started

be connected. If there is any trouble connecting to the server, or if your login name has already been taken, you will receive a message in the status bar, informing you that you have to change the server address or your name and try to connect again. If you are successfully connected, you will receive a welcome message from the server.

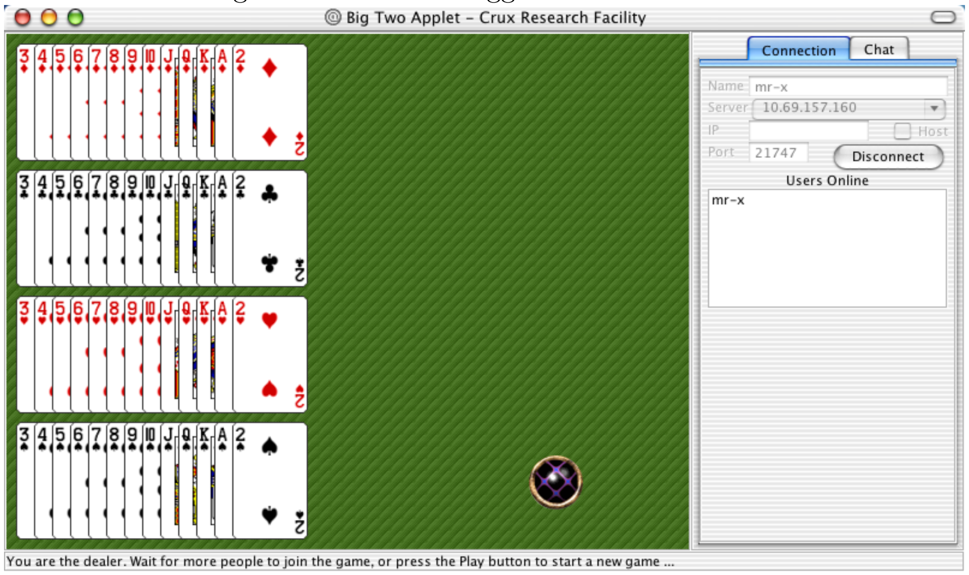
2.5 Playing the Game

If you are the first person entering the game, you are the dealer. A purplish button will appear on the screen (See Fig. 2.5; this will be the "play button" later on for the game too). You can wait for more players to join the game, or you can click on the button to begin the game with existing players. The computers will take the empty seats. If there are four players logged in, the game will begin automatically.

If you are not the first one to login the game, you will not see the button. It's either that the dealer has not started the game yet, or there is an ongoing game. In the latter case, you will be able to join the game as soon as the current round is finished, as long as there are any empty seats (a computer will vacate its seat for you).

As for the game itself, it goes counter-clockwise. For the first game or for the game after there is a change in players, the player who has the three

Figure 2.5: You are logged in as the dealer.



Name: mr-x
Server: 10.69.157.160
Port: 21747
Users Online: mr-x

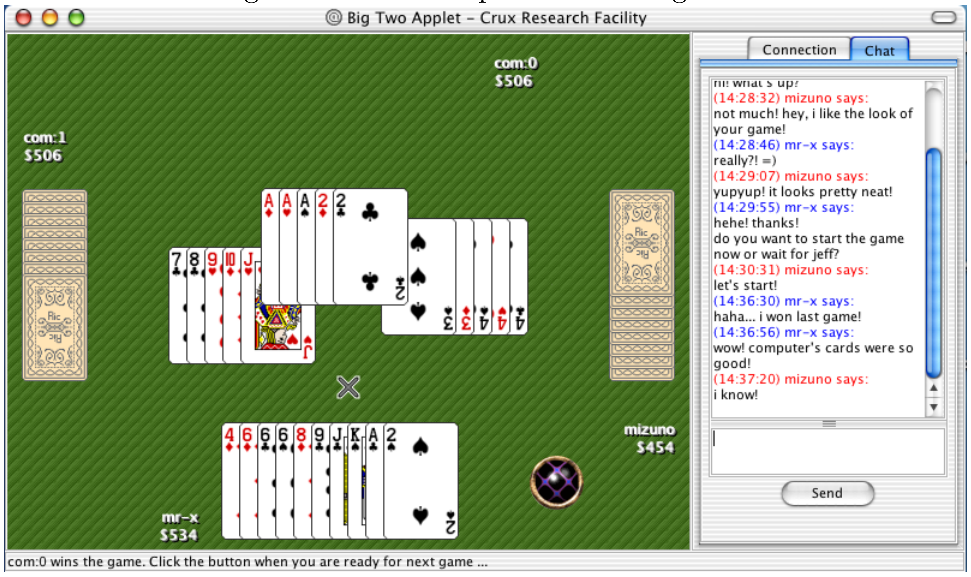
You are the dealer. Wait for more people to join the game, or press the Play button to start a new game ...

of diamonds plays first. For all successive games, the winner plays first. The name of the person becomes orange when it's his or her turn to play. You can select or de-select your cards by clicking on them. To play your selection, click on the button, and your selection will appear in front of you. You will get a warning if your selection is illegal and you will have to re-select your cards. To pass, simply not select any cards and click on the button, and it will be the next player's turn. Instead of showing your cards on the spot where you place them, there will be a big $\times$. You will get a warning in the status bar if a player has only one card left. If someone wins the game, you can click on the button when you are ready for the next game (Fig. 2.6). When all players are ready, a new game will begin.

2.6 Chatting

If you are playing with other users and wish to chat with them, click on the "Chat" tab. To type in a message, enter the text in the bottom pane (Fig. 2.7) and hit `Return` or `Enter`. You can also click on the "Send" button to send a message. To input a line break within your message, hit `Control-Return` or `Shift-Return`.

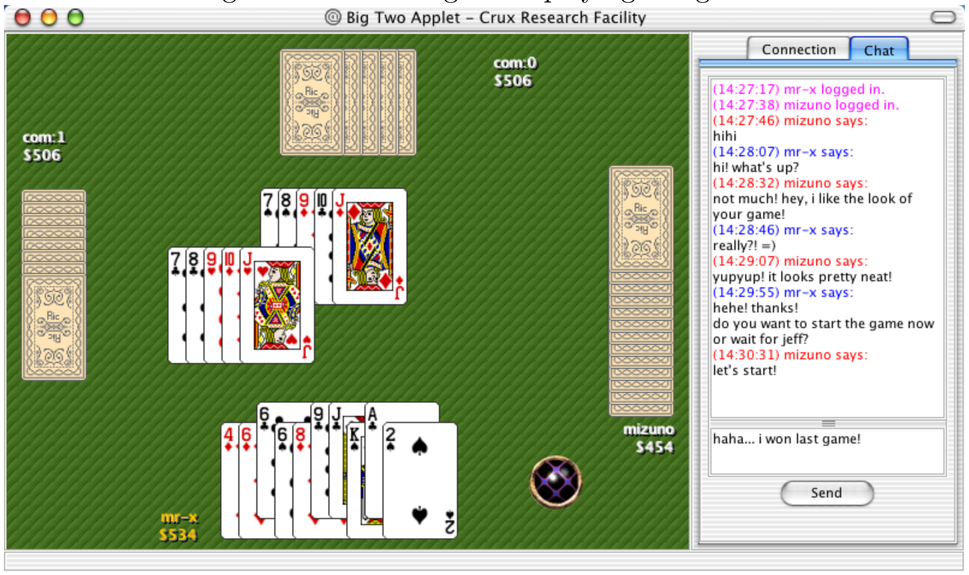
Figure 2.6: The computer wins the game.



(14:28:32) mizuno says:
not much! hey, i like the look of your game!
(14:28:46) mr-x says:
really?! =)
(14:29:07) mizuno says:
yupyup! it looks pretty neat!
(14:29:55) mr-x says:
hehe! thanks!
do you want to start the game now or wait for jeff?
(14:30:31) mizuno says:
let's start!
(14:36:30) mr-x says:
haha... i won last game!
(14:36:56) mr-x says:
wow! computer's cards were so good!
(14:37:20) mizuno says:
i know!

com:0 wins the game. Click the button when you are ready for next game ...

Figure 2.7: Chatting while playing the game.



(14:27:17) mr-x logged in.
(14:27:38) mizuno logged in.
(14:27:46) mizuno says:
hihi
(14:28:07) mr-x says:
hi! what's up?
(14:28:32) mizuno says:
not much! hey, i like the look of your game!
(14:28:46) mr-x says:
really?! =)
(14:29:07) mizuno says:
yupyup! it looks pretty neat!
(14:29:55) mr-x says:
hehe! thanks!
do you want to start the game now or wait for jeff?
(14:30:31) mizuno says:
let's start!

haha... i won last game!

2.7 Logging Out

If you have to leave the game, click on the "Disconnect" button in the "Connection" tab. If there is an ongoing game, the game will be interrupted and a computer player will replace you. If you are the only human player in the game and you log out, then the game is over. The server will be there waiting for a new dealer.

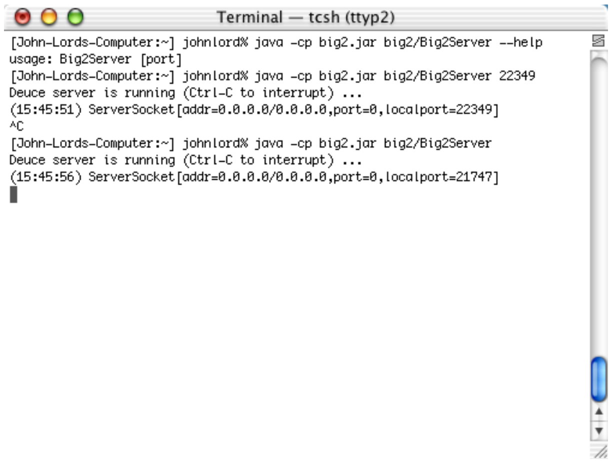
2.8 Hosting

To host the game, you have to download a file called `big2.jar`. The link is provided on the Deuce Web page. After the downloading is done, in a terminal[2], change to the directory to where you downloaded the file and type in the following:

```
java -cp big2.jar big2/Big2Server [port]
```

The port is optional. If you leave it out, the standard Deuce port will be used. The server will then be up and running (Fig. 2.8). You can obtain

Figure 2.8: The server is up and running in a terminal.



```
[John-Lords-Computer:~] johnlord% java -cp big2.jar big2/Big2Server --help
usage: Big2Server [port]
[John-Lords-Computer:~] johnlord% java -cp big2.jar big2/Big2Server 22349
Deuce server is running (Ctrl-C to interrupt) ...
(15:45:51) ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=22349]
^C
[John-Lords-Computer:~] johnlord% java -cp big2.jar big2/Big2Server
Deuce server is running (Ctrl-C to interrupt) ...
(15:45:56) ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=21747]
```

your IP address[3] from the Deuce Web page. You can then tell your friends that you are hosting the game and they can connect to the server you are running.

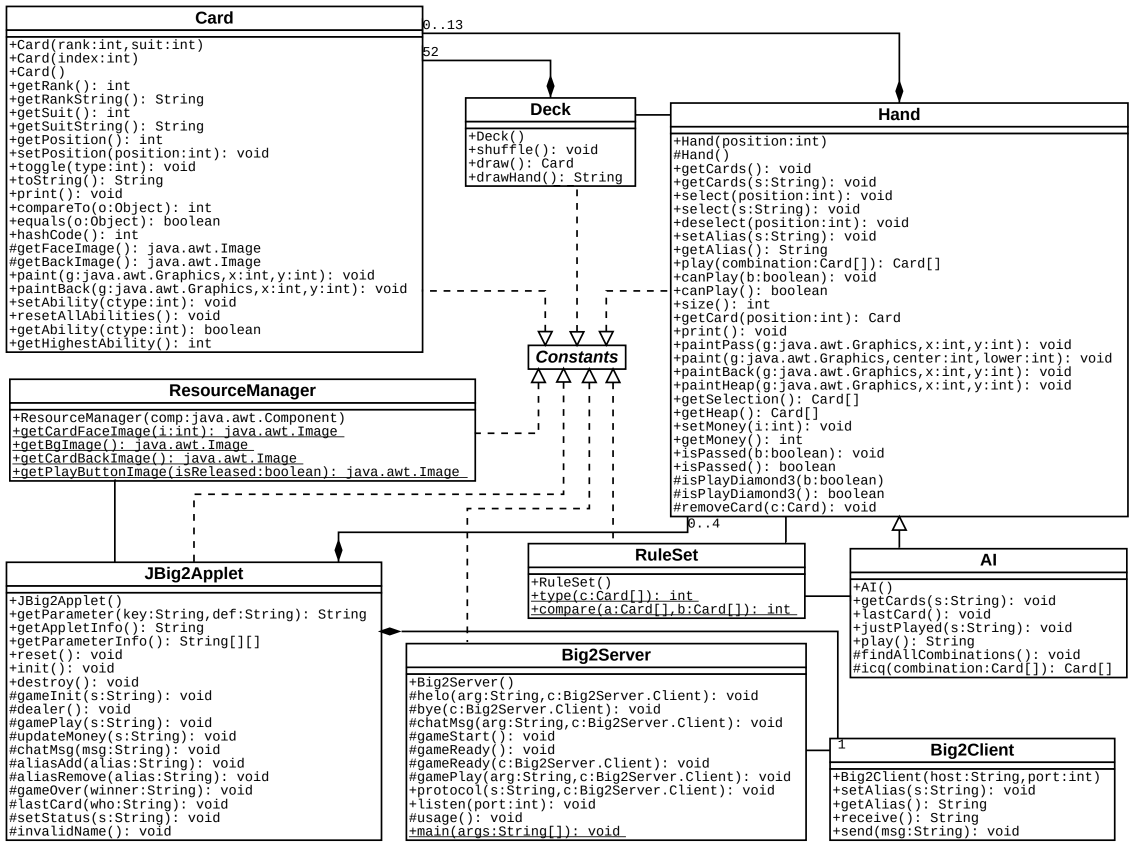
[2]For Windows, in *Command prompt* or *MS-DOS*

[3]If you are behind a firewall or an IP-masking program, your IP address may not be correctly shown and/or certain ports may not be open to you. Contact your local system administrator for instructions.

Chapter 3

The Design

3.1 The UML Diagram



Card
+Card(rank:int,suit:int)
+Card(index:int)
+Card()
+getRank(): int
+getRankString(): String
+getSuit(): int
+getSuitString(): String
+getPosition(): int
+setPosition(position:int): void
+toggle(type:int): void
+toString(): String
+print(): void
+compareTo(o:Object): int
+equals(o:Object): boolean
+hashCode(): int
#getFaceImage(): java.awt.Image
#getBackImage(): java.awt.Image
+paint(g:java.awt.Graphics,x:int,y:int): void
+paintBack(g:java.awt.Graphics,x:int,y:int): void
+setAbility(ctype:int): void
+resetAllAbilities(): void
+getAbility(ctype:int): boolean
+getHighestAbility(): int
0..13
52
Deck
+Deck()
+shuffle(): void
+draw(): Card
+drawHand(): String
Hand
+Hand(position:int)
#Hand()
+getCards(): void
+getCards(s:String): void
+select(position:int): void
+select(s:String): void
+deselect(position:int): void
+setAlias(s:String): void
+getAlias(): String
+play(combination:Card[]): Card[]
+canPlay(b:boolean): void
+canPlay(): boolean
+size(): int
+getCard(position:int): Card
+print(): void
+paintPass(g:java.awt.Graphics,x:int,y:int): void
+paint(g:java.awt.Graphics,center:int,lower:int): void
+paintBack(g:java.awt.Graphics,x:int,y:int): void
+paintHeap(g:java.awt.Graphics,x:int,y:int): void
+getSelection(): Card[]
+getHeap(): Card[]
+setMoney(i:int): void
+getMoney(): int
+isPassed(b:boolean): void
+isPassed(): boolean
#isPlayDiamond3(b:boolean)
#isPlayDiamond3(): boolean
#removeCard(c:Card): void
Constants
ResourceManager
+ResourceManager(comp:java.awt.Component)
+getCardFaceImage(i:int): java.awt.Image
+getBgImage(): java.awt.Image
+getCardBackImage(): java.awt.Image
+getPlayButtonImage(isReleased:boolean): java.awt.Image
0..4
JBig2Applet
+JBig2Applet()
+getParameter(key:String,def:String): String
+getAppletInfo(): String
+getParameterInfo(): String[][]
+reset(): void
+init(): void
+destroy(): void
#gameInit(s:String): void
#dealer(): void
#gamePlay(s:String): void
#updateMoney(s:String): void
#chatMsg(msg:String): void
#aliasAdd(alias:String): void
#aliasRemove(alias:String): void
#gameOver(winner:String): void
#lastCard(who:String): void
#setStatus(s:String): void
#invalidName(): void
RuleSet
+RuleSet()
+type(c:Card[]): int
+compare(a:Card[],b:Card[]): int
AI
+AI()
+getCards(s:String): void
+lastCard(): void
+justPlayed(s:String): void
+play(): String
#findAllCombinations(): void
#icq(combination:Card[]): Card[]
Big2Server
+Big2Server()
#helo(arg:String,c:Big2Server.Client): void
#bye(c:Big2Server.Client): void
#chatMsg(arg:String,c:Big2Server.Client): void
#gameStart(): void
#gameReady(): void
#gameReady(c:Big2Server.Client): void
#gamePlay(arg:String,c:Big2Server.Client): void
+protocol(s:String,c:Big2Server.Client): void
+listen(port:int): void
#usage(): void
+main(args:String[]): void
1
Big2Client
+Big2Client(host:String,port:int)
+setAlias(s:String): void
+getAlias(): String
+receive(): String
+send(msg:String): void

3.2 Objects

The genesis of the computer revolution was in a machine. The genesis of our programming languages thus tends to look like that machine. But computers are not so much machines as they are mind amplification tools ("bicycles for the mind," as Steve Jobs is fond of saying) and a different kind of expressive medium. As a result, the tools are beginning to look less like the machines and more like parts of our minds, and also like other forms of expression such as writing, painting, sculpture, animation, and filmmaking. Object-oriented programming (OOP) is part of this movement toward using the computer as an expressive medium.

— Bruce Eckel, Thinking in Java 2nd Edition, p. 29.

This game takes full advantage of object-oriented programming. As a pure object-oriented language, Java makes the development of the game more like riding the "bicycle for the mind." Everything is an object in Java, and this makes coding Big Two so much more intuitive. A `Card` is an object; a `Hand` (of cards) is an object; a `Deck` is an object; and you, a `Client`, are also an object. The objects by themselves don't do much; it's the interaction of the objects which makes the program interesting and sophisticated. Different combinations of cards can have different values, where some are legal and others are not. A `Client` object at the server side keeps track of all remote clients; it keeps their names, keeps connections with their computers, and keeps interacting with the program at the client side. Object is the building block of this game. Without it, it would be much harder to make the game, and even harder to maintain it.

3.3 Inheritance

Inheritance is the second most essential feature in object-oriented programming language, after data abstraction.

Inheritance is an object-oriented trick to use the classes without soiling the existing code. It creates a new class as a *type* of an existing class. For example, in the early stage of the project, I created the `Hand` class. Below is a snippet of the code (the actual code is much longer):

```
public class Hand {
    /**
```

```
     * Constructs a hand with position information.
     * @param position  the position of the player, such as
     *        <code>NORTH</code> or <code>EAST</code>.
     */
    public Hand(int position) {
        // ...
    }

    /**
     * Draws 13 cards from the deck and sorts them.
     * @param s  The string representation of the hand, for
     *           example, <code>
     *           "45 23 1 2 34 5 3 42 37 12 23 10 9"</code>.
     */
    public void getCards(String s) {
        // ...
    }

    public void setAlias(String s) {
        // ...
    }

    /**
     * Selects a card to be played either alone or with other selected
     * cards; de-selects this card if it has been selected
     * previously.
     * @param position  the position in the hand of the card
     *                  to be selected/de-selected
     */
    public void select(String s) {
        // ...
    }

    /**
     * Plays the selected cards.
     * @return  a combination of cards played if they are legal to play;
     *          <code>null</code> if they are illegal
     */
    public Card[] play(Card[] combination) {
        // ...
    }
    // ...
}
```

This class represents a hand of cards; it contains essential methods to manipulate the hand, such as drawing cards from a deck, selecting or de-selecting cards, and playing the selection. The methods are designed to be interactive with the users. For example, when a user clicks on a card, a `mouseClicked` event is captured. The co-ordinates are then translated into the corresponding card and the `select` method is called.

Now, later on in this project, I decided to start making computer players, or the so-called *artificial intelligence* (AI). The AI is an algorithm[1] which makes a hand play cards automatically. The combination of cards played by the AI depends on several factors, such as how many cards the opponents have left. So essentially, an `AI` *is* a type of `Hand`, a hand that can think. So there is no need to re-create a whole new class, most of the methods in `Hand` can still apply to `AI`. That's when inheritance comes to play.

Essentially, all I have to do is to change the `play` method, and add methods that perform the algorithms. Here is a snippet of the `AI.java`:

```
public class AI extends Hand {
    /**
     * Overrides the same method in the base class. Performs
     * extra steps to initialize the AI.
     */
    public void getCards(String s) {
        // ...
    }
    protected void findAllCombinations() {
        // ...
    }
    /**
     * Calls the algorithm methods and plays appropriate cards.
     */
    public String play() {
        // ...
    }
    // ...
}
```

[1]The algorithm is described in Section 3.5.

There is no need to rewrite the methods that are already included in `Hand`. Thus inheritance greatly simplified the process of reusing a class. In the next section, we will see that inheritance also makes the existing class very expandable through the use of polymorphism.

3.4 Polymorphism

Polymorphism provides another dimension of separating interface from implementation. It allows improved code organization and readability as well as the creation of extensible programs that can be “grown” not only during the original creation of the project, but also when new features are desired. The use of polymorphism is described using an example from this project. The `Big2Server` has an inner class called `Client`. Whenever a player connects to the server, an instance of `Client` in `Big2server` is created. The `Client` class has a method for receiving a message from a client (`receive`) and a method for sending a message (`send`). Whenever the server receives a message from a client, it calls a method called `protocol` to parse the message and broadcast the message to all or some of the clients connected, depending on what the message is.

Now, the AI gets involved again. The `Client` cannot be used for AI since the AI is not a remote client and it is not connected to a socket. I created a new inner class called `AIClient` which is inherited from `Client`. I overrode the `send` and `receive` methods in the base class. The overridden methods manipulate an `AI` object, as described in the last section, to make the AI play cards automatically.

However, the server sends a message through `protocol`, so how does it know whether to send a message to a human player through a `Client` or to send a message to a human through an `AIClient`? It doesn’t. Since `protocol` was written before the AI, it seems that I have to change the code to make the protocol talk to the AI. I could use some “if” clauses to test whether it is to broadcast to a `Client` or to a `AIClient`, but I don’t have to. By using polymorphism, I ended up not having to change a single line of `protocol`.

Since `AIClient` *is* a `Client`, it makes sense that when I can talk to a `Client`, I can talk to any types of `Client`, including `AIClient`. This is due to the methods in the base class being all available in the inherited classes. So I instantiated an `AIClient` as such:

```
Client ac = new AIClient();
```

The `AIClient` is *upcasted* to a `Client`. Thus all the code in `protocol` that talks to a `Client` can also apply to `AIClient`. And the smart part of polymorphism? It can recognize the inherited classes and call the methods that override the ones in the base class, so the new `send` and `receive` methods are being called successfully.

3.5 Artificial Intelligence

The artificial intelligence in this game is just a fancy name for a set of algorithms that enable the computer to play cards automatically. The algorithm used so far in this game is a not-so-flexible one that basically does seeking and finding. The algorithm is described below.

Whenever an opponent plays a combination, the AI gets notified. The AI remembers the last combinations played by the opponents. When it is its turn, it first calls a method named `findAllCombinations`. Basically what this method does is to search the whole hand and find all possible combinations, and save all those combinations to the appropriate place in an array. Remember that in Java, objects are copied by their references, so no new objects are created in this process; just their references are being copied. When a new combination is found, the “ability” information is also added to an individual card. For example, the AI has a hand of six cards left, ♢8-♠8-♢K-♣K-♡K-♠K. After calling the method `findAllCombinations`, ♠K will have the abilities of `SINGLE`, `PAIR`, `TRIPLE`, `FULL_HOUSE`, and `QUADS` (the capitalized words are integral constants defined in the class).

Then the computer takes a look at the combination played by others. If all three opponents have passed, the computer decides what cards to play and begins a new round; otherwise, the computer searches through the combination list to find a combination to beat the previous play, or, the computer can pass.

Whether to play a combination or not mainly depends on the abilities of each card in the combination. For example, the computer has, again, the following hand: ♢8-♠8-♢K-♣K-♡K-♠K. Now an opponent plays ♣J-♡J. The computer searches through the combination list and finds ♢K-♣K, which is higher than ♣J-♡J. But is the computer going to play it? No. The computer examines each card, say ♢K, and finds that its highest ability is four-of-a-kind. If it plays the pair, then it would break such a good hand.

However, the abilities of a card are not the only factors to determine whether to play a card or not. Factors, such as the remaining cards left in other players’ hands, also affect the computer’s decision. The computer would also take chances, just like any human being, by the means of a probability random number generator. For example, a computer finds an appropriate pair to beat the previous one; it has a, say, 90 % chance to play the pair. The following lines are used to achieve this:

```
if(play && Math.random() < 0.9)
    return c; // c is the array containing the pair
```

Note that when the computer decides to play the pair, the boolean flag `play` becomes `true`.

Although the ability of this algorithm is limited, the methods used, such as the random number generator, make the computer player less predictable and make the game more enjoyable when the computer players are involved.

Appendix A ChangeLog

```
2003-01-15  xc  <cx@sdf>

* AI.java, Big2Client.java, Big2Server.java, Card.java,
* Constants.java, Deck.java, Hand.java, JBig2Applet.java,
* ResourceManager.java, RuleSet.java:
Finished documentation. The game has reached a stable
stage.

2003-01-13  xc  <cx@sdf>

* Big2Server.java, JBig2Applet.java:
Fixed the bug which occurs when users login as the game is
playing.  Fixed the bug that garbled the chat text-pane.

2003-01-12  xc  <cx@sdf>

* AI.java, Big2Client.java, Big2Server.java, Hand.java,
* JBig2Applet.java:
Completed login and logout support. When a user logs in,
he/she will replace a computer player; when a user logs
out, a computer player will replace him/her.  Now the
server is truly a ‘‘server’’ since theoretically it’s not
nessesary to shut it down because you need more players
when a player has left.  The server can always be on.

The login name verification has also been improved. If you
login using a name that has already been taken, you will
be disconnected and informed to use another name.

The server command line message now features a time-stamp.
```

```
Fixed the bug involving the AI not cleaning up its record
about the last played combination when a new game starts.

2003-01-09  xc  <cx@sdf>

* AI.java, Big2Server.java, Constants.java,
* JBig2Applet.java:
Improved the disconnecting function. Made the AI smarter.
Finished the login name restriction.  Fixed several minor
bugs.

2003-01-02  xc  <cx@sdf>

* Big2Server.java, Card.java, Constants.java, Hand.java,
* JBig2Applet.java, AI.java:
Added artificial intelligence. Now one can play with three
other players, or three computers, or a mix of network
players and computers.

2002-12-25  xc  <cx@sdf>

* Big2Server.java, Constants.java, Hand.java,
* JBig2Applet.java, RuleSet.java:
The game is playable with 4 network players, and with some
nice features (such as playing with money). All major bugs
are fixed. The game can now go on after the first round.
Reached a milestone (playable).

2002-12-18  xc  <cx@sdf>

* Hand.java:
Fixed the problem involving the ‘‘remaining number of
cards’’ (hopefully).

2002-12-17  xc  <cx@sdf>

* Big2Server.java, Deck.java, Hand.java, JBig2Applet.java:
Finished the ‘‘play cards’’ stuff. Will reach a milestone
soon. All logics are working fine under very limited
```

```
testing. One can select cards to play (with other
networked players) or choose to pass (of course you cannot
pass if the other three have passed). The implementation
for winning is not done yet (though it should be pretty
easy).  To do: place the word ‘‘pass’’ on the table when
someone passes.

2002-12-12  xc  <cx@sdf>

* Big2Client.java, Big2Server.java, Constants.java,
* Deck.java, Hand.java, JBig2Applet.java,
* ResourceManager.java:
Finished the layout of the heap of cards in the middle of
the table.  The code for the position of the combination
for different sides of the table is in Hand.java.

2002-12-08  xc  <cx@sdf>

* Constants.java, Hand.java, JBig2Applet.java,
* ResourceManager.java:
The player can now select cards. The selected cards will
be raised.  There is a clickable button to the right which
plays the selection.  The implementation of the button has
not yet been done.

* Big2Server.java, Card.java, Constants.java, Hand.java,
* JBig2Applet.java:
Server now can handle logout. When a player logs out, his
name is removed, and the connection is closed; other
players are notified about the logout.

The ‘‘show cards’’ part is finished. After 4 people have
logged into the game, you can immediately see your cards,
as well as the back of others’ cards at the other three
sides of the table.

2002-11-21  xc  <cx@sdf>

* Constants.java:
Resized the cards. Modified the cards so now they look
```

```
like real playing cards.

2002-11-20  xc  <cx@sdf>

* Big2Server.java, Card.java, Constants.java, Deck.java,
* JBig2Applet.java, ResourceManager.java:
Added better image support. Used a new class called
ResourceManager to load all the images first and return
the reference of the appropriate image when necessary.
The server can deal cards and the client can display the
cards in a hand.  Implementations have not yet been put
together. Card images need to be resized (it's too big, I
will resize them to maybe to 0.8 of the original).

2002-11-15  xc  <cx@sdf>

* Constants.java, JBig2Applet.java:
Fixed all known bugs so far. The chat is now fully
functional, including a multi-line support. Users can now
press RETURN anywhere in the textArea to send a message
(without breaking a line) or users can press SHIFT-RETURN
or CTRL-RETURN to get a new linefeed (just like what many
popular chat clients do nowadays).

2002-11-14  xc  <cx@sdf>

* JBig2Applet.java:
Fixed all major bugs in chat. The ``welcome bug'' still
exists. Added more functionalities to the chat such as a
timestamp for each message.

2002-11-13  xc  <cx@sdf>

* JBig2Applet.java:
Added a welcome message in the chat which is displayed
when a user logs in. Pressing the enter key in chat will
now send the message. The auto-scroll also worked but it
is so buggy in Windows systems that people cannot even see
the messages, so this feature is still temporary disabled.
```

```
2002-11-12  xc  <cx@sdf>

* Big2Server.java, JBig2Applet.java, Big2Client.java:
Improved the chat with the following: a new color scheme
and an online notification. Disconnecting still does not
work. Still have to fix the multiple-line problem and the
auto-scroll problem.

2002-11-06  xc  <cx@sdf>

* Big2Client.java, Big2Server.java:
Network client and server for Big Two. Kind of working
with (very) limited features.

* Constants.java, JBig2Applet.java:
Added network capability. Not done yet.

* Card.java: Small bug fix.

2002-10-11  xc  <cx@sdf>

* JBig2Applet.java:
Added interface to the connection tab. No implementation
has been done.

2002-10-09  xc  <cx@sdf>

* JBig2Applet.java:
Packed into a JAR file. The images load successfully now.
Changed chatTextArea to chatTextPane to allow rich text.

2002-10-08  xc  <cx@sdf>

* Card.java, Constants.java, Deck.java, JBig2Applet.java,
* RuleSet.java:
Renamed CardConstants.java to Constants.java. Files are
modified according to the new class name.

* CardConstants.java, Constants.java:
Renamed CardConstants.java to Constants.java
```

```
2002-10-04  xc  <cx@sdf>

* JBig2Applet.java:
New file. It is the GUI part of the game. Done very basic
layout.

* Card.java, CardConstants.java, Deck.java, RuleSet.java:
Retabbed files

2002-09-26  xc  <cx@sdf>

* Deck.java: Added @throws tag to the doc comment of the
* method draw()

* Card.java, Deck.java, RuleSet.java: Reformatted all doc
* comments.

2002-09-24  xc  <cx@sdf>

* Card.java: Fixed a bug in main()

* RuleSet.java:
Modified the test code in main(). Each deck is shuffled
before any Cards are drawn. (In the old Deck class, it was
not necessary.)

* Deck.java:
The field deck inside the class is changed from an
ArrayList to a Card[], and since in draw() method the
Cards are no longer removed from deck, the Card objects
are reusable in each round.

* CardConstants.java:
Changed the values of the constants DIAMOND, CLUB, HEART,
and SPADE to 0, 1, 2, and 3, respectively.

* Card.java:
Changed the range of the field ‘index’ (the value of the
index is returned by hashCode()) from [1, 52] to [0, 51].
```

```
Now a Card with an index 0 represents three of diamonds;
and a Card with a index 51 represents two of spades.

2002-09-23  xc  <cx@sdf>

* Card.java, Deck.java, RuleSet.java:
Rearranged the sequence of fields and methods in the
classes.

* Deck.java:
Updated method play() in inner class Hand to suit the
change in RuleSet.  play() now returns a Card array.

* RuleSet.java:
All static methods in RuleSet now takes a Card[] parameter
instead of an ArrayList. Doing this is for efficiency and
design. There is actually no need for an ArrayList here
since resizing (adding or removing) is not necessary.

2002-09-22  xc  <cx@sdf>

* RuleSet.java: Fixed some typo in comment documentation.

* Deck.java:
Replaced the private method sort() in the inner class Hand
with updatePosition(). Sorting a Hand of Cards is only
necessary in the constructor.  There is no need to include
it in a method. However, after each time the hand is
modified, the position of the Cards must be updated.

2002-09-21  xc  <cx@sdf>

* Deck.java: Added comments to the Hand inner class.

* Card.java, CardConstants.java, Deck.java, RuleSet.java:
Imported Big Two sources

* Card.java, CardConstants.java, Deck.java, RuleSet.java:
* New file.
```

Appendix B Resources

`http://cx.freeshell.org/big2/index.php`	Deuce main page
`http://cx.freeshell.org/big2/src/`	Source code
`http://cx.freeshell.org/big2/ChangeLog`	The ChangeLog
`http://cx.freeshell.org/big2/doc/`	The Deuce API
`http://cx.freeshell.org/big2/faq.php`	Frequently asked questions

Index