MUDIT AGARWAL
RYAN KENNEDY

UNIVERSITY OF NOTRE DAME

16 DECEMBER 2002

FINAL PROJECT

GUI BASED BLACKJACK USING EIFFEL 5.2

FINAL PROJECT

GUI BASED BLACKJACK USING EIFFEL 5.2

ABSTRACT

Our goal was to use EiffelStudio 5.2 to recreate the famous card game BlackJack. We wanted to use as many data structures that we learned in class as possible in accomplishing this. We used data structures to represent the dealer, player, cards, banker, and shuffler. To our knowledge, BlackJack had not been previously created in EiffelStudio, so our main problem was that we had little to refer to when creating the game. We utilized EiffelStudio's core integration of design by contract when creating our program. EiffelStudio is a very pleasant environment and it is likely that many more applications will be made using it in the future.

KEYWORDS

1. Artificial Intelligence: dealer
2. BlackJack: famous card game; also called 21
3. Classes: form outlining structure
4. Gambling: betting chips
5. Graphical User Interface: colorful, easy to use interface
6. Hash Table: quick method of relating a key to its value
7. Inheritance: ability to reuse classes by absorbing their attributes and behaviors
8. Multiplayer: support for multiple players
9. Polymorphism: enables us to write programs in a general fashion to handle a wide variety of existing and yet-to-be-specified related classes
10. Random Function: use of random number generator
11. Two-Way Circular List: data structure which allows you to traverse the list to the end and then back to the top

INTRODUCTION

We were determined to create the popular BlackJack game in EiffelStudio because we were unable to find any previous similar usage of EiffelStudio. We could not find any references to this or any other card game in EiffelStudio and we wanted to lay some groundwork for future work in this area. Our strategy was to first implement a text-based version of this game through the use of many classes and then to extend this to a graphical user interface with rich images. Our work was mostly trial and error because we were also unable to find any references for the extensive amount of images that we used. We did initially refer to Yahoo!'s version of BlackJack, but wanted to create a unique version of our own, complete with custom images and interface. We feel that we were very successful in accomplishing our goal.

BODY

GAMEPLAY

Our game was designed to be very easy to play. The game follows the standard BlackJack rules which state that goal is to get your cards to sum as close to 21 as possible, without going over. Being initially dealt 2 cards which sum to 21 constitutes a BlackJack. Face cards are worth 10, aces are worth 11 unless that pushes the sum over 21, in which case their value is changed to 1, and all other cards are worth their face value. The user can bet a specified number of chips for each hand and quit at the completion of a hand. Players begin with 2 cards and can take as many hits as they would like so long as their total is less than 21. A hit constitutes a new card being dealt to the user. The dealer begins with 2 cards, only 1 of which is known by the user. The dealer keeps taking hits until his sum is greater than or equal to 17, unless the user busts.

TEXT-BASED GAME

We set about creating our game in steps. Our first goal was to lay the foundations of the graphical BlackJack by creating a text-based version. A diagram for our class structure can be found in Figure 1. The most important classes were the following:

Main (not submitted):	tests our game.
Random_seeder:	creates random seeds for shuffling the cards in a different way every time, utilizing the time class
Blackjack_dealer:	tells the dealer when to hit and when to stand (follows *Soft 17* format)
Human_blackjack_player:	asks the player to choose between hit and stand; also asks for bets
Hand:	stores the player's cards
Chips:	adds and removes chips from the player's wallet

We found it easiest to incorporate a number of data structures into our project. We implemented both data structures taught in class and some we learned on our own to make the programming easier for us. The most noteworthy data structures used were the following:

Two-Way Circular List: This was used to store the list of players. It was particularly useful in the text-based version that supported more than one player and would make it much easier to upgrade our GUI to have multiplayer support. We used this data structure to keep the pointer on the proper player for the duration of his turn and for the dealing of the cards. After the player it is pointing to finishes his turn, it moves to the next player in the list.

Hash Table: We used a hash table to keep track of the player's name and the number of chips he has. This was particularly useful in the text-based version and makes it easier for us to expand the GUI version to a multiplayer game.

Time Class: To get the random seed to shuffle the cards, we used EiffelStudio's included time class. This proved to be difficult to implement because we were required to convert the time to integer format. This was done by converting the time first to a string and then that string to a hash code. The command string.hash_code gives the hash code of the string.

Playing Card Constants: These were utilized to define the values of all the cards as per their value and rank. Each card pointed to its corresponding image.

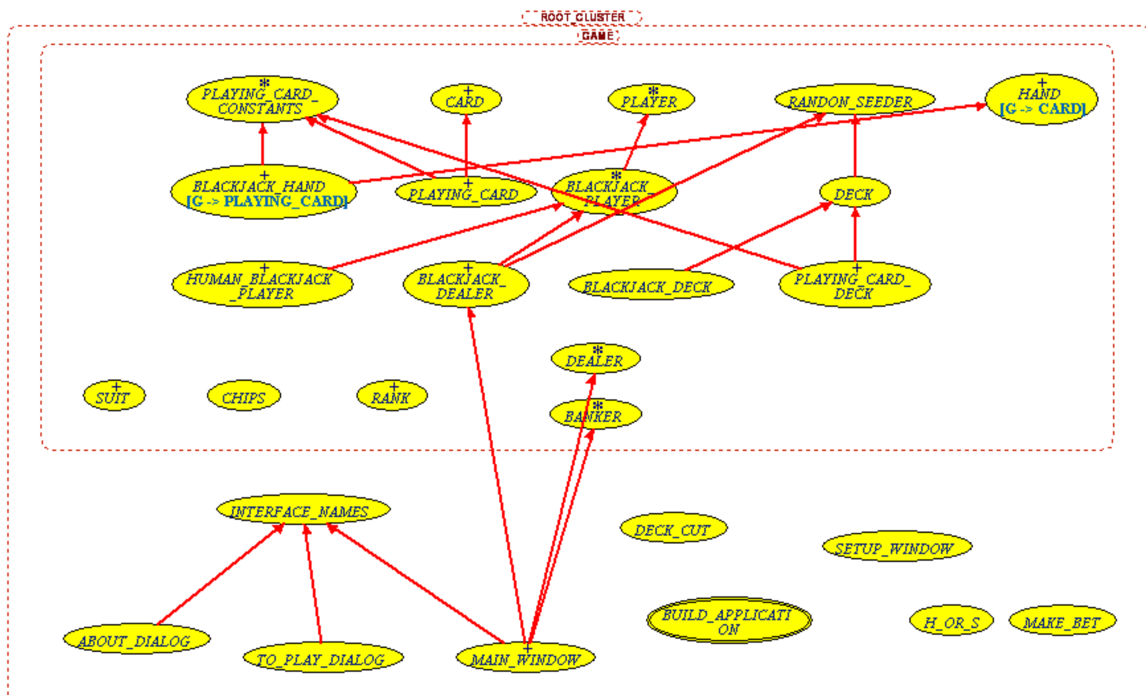

Figure 1

GRAPHICS AND PLAYING THE GAME

Next, we created the graphics and implemented them into a GUI. The basics for the deck of cards were lifted from Microsoft ® Solitaire and edited through JASC ® Paint Shop Pro ™. The face cards and card backs were most thoroughly edited, often implemented with images from http://www.nd.edu. The main background was adapted from http://www.casinoonet.com and the chips from http://www.pokerchips.com. We added our own personal flair to each of the graphics. All graphics were converted to the Portable Network Graphics (.png) format. Utilizing the EV_FIXED class from EiffelStudio, we were able to create EV_PIXMAPs to display each image. We used EV_FIXED because it was the only way we could figure out how to "stack" images. Each of our images were implemented as EV_BUTTONs because buttons were the easiest to manipulate.

EiffelBuild generated our basic layout; however, we made many changes and additions. We included a *How To Play* EV_DIALOG box in addition to customizing the *About* dialog box and file menu. The EV_PIXMAPs have been customized in the dialog boxes as well. We included a confirmation box to confirm the user's decision to exit and have set the dimensions to fit our images accordingly. Among other widgets and primitives used were:

EV_TITLED_WINDOW, EV_TEXT, EV_TEXT_FIELD, EV_LIST, EV_VERTICAL_BOX, EV_HORIZONTAL_BOX, and EV_ENVIRONMENT.

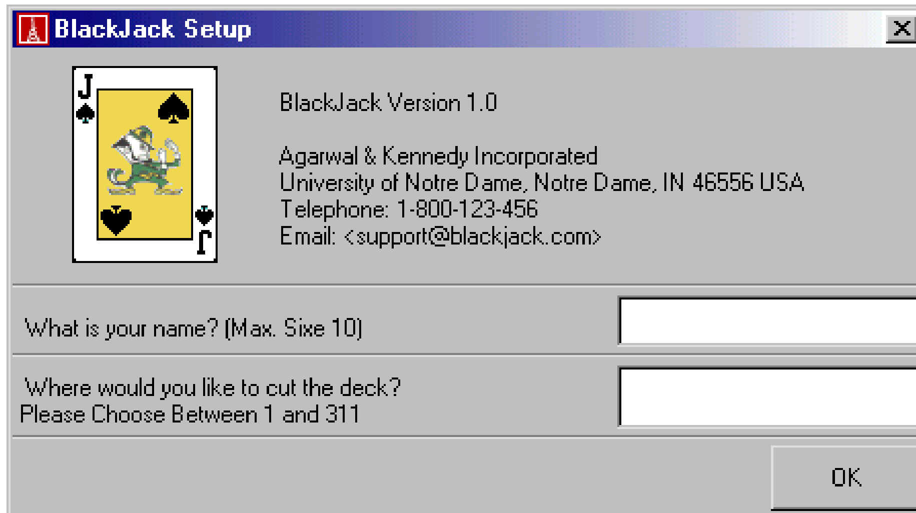

Figure 2

We designed the game to begin by opening a setup window, Figure 2, in which the user enters both his name and where he wants to cut the deck. We have used contracts to ensure that the program won't function unless the name text field contains some characters and the cut deck text field contains a valid number (as specified in the window). Our use of contracts makes sure that a valid number was entered. Once both conditions have been satisfied, the user clicks *OK* and is taken to the next window. The user then clicks *Start Game* and the main window, Figure 3, appears.

The main window contains many user-friendly features. First, the user's name is displayed under his hand and his total number of chips to the left of that. Second, the current total for the user's hand is displayed to the right of his hand. Lastly, the end total (or bust) of the dealer is displayed just above the user's total. We have also included a stack of cards and chips to add to the overall feel of the game.

The user sets his bet in the set bet window, Figure 4, by clicking on his selection from the list. A user can bet 1, 5, 10, or 25 chips and begins with 5000 chips. If no bet is selected, we ensure that 1 chip is bet by default. The player next presses the *Deal* button and the cards are dealt. A window, Figure 5, then pops up in which the user decides whether to hit or stand. After pressing the button of his choice, he is taken back to the main window where he must click *OK* before the next card is dealt. This continues until he clicks *Stand* or until he busts. When *Stand* has been pressed, the user must again click *OK* and then the dealer's card is finally turned over and his additional cards (if any) dealt. Depending on the outcome, a *You Win*, Figure 6, or *You Lose*, Figure 7, appears which the user clicks to continue to the next hand. If the player busts, the *You Lose* button appears and the dealer's second card is overturned.

Figure 3

Figure 4

Figure 5

Figure 6

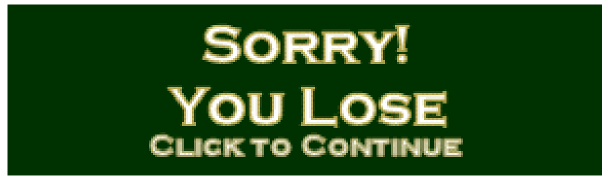

Figure 7

LINKING

Linking the text-based game to the GUI required more work than we had anticipated. The modifications we made caused the program to no longer support text-based play, as all *io* commands were removed. We did not know how to take a value from one window and pass that value to another window, so we had to implement a button mechanism that updated the values between windows. Hence, we have a lot of buttons that move ahead in the program. For example, when the user enters his name and where he would like to cut the deck, he has to hit *OK* to exit that window and then *Start Game* once he has been taken to the main window. The *Start Game* button allows the main window to get the information entered in the setup box. This proved to be very difficult to figure out, as we could find no documentation.

Another difficulty we encountered involved the *io.put_string* command. In the text-based version, we could use this in any class, but in the GUI we could not. We were forced to make the hit or stand window local to the main window class because the hit or stand *io* command used in the text-based version did not work. This proved to be somewhat good in the end, as we had linked all the classes and used polymorphism and inheritance. We were able to use almost everything we were taught in class. The final structure can be seen in Figure 8.

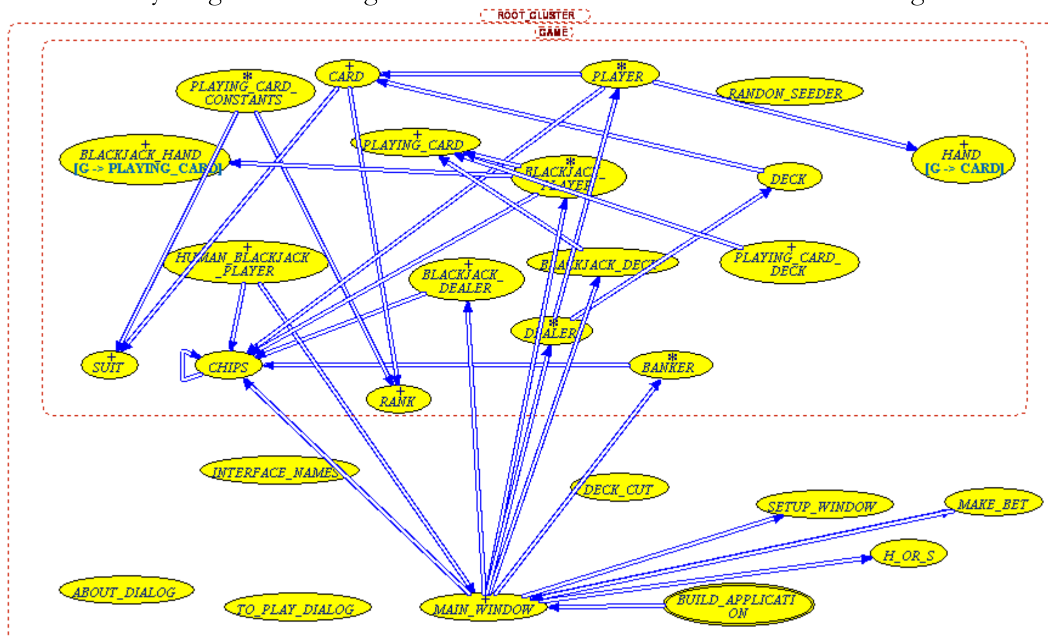

Figure 8

SUMMARY

In conclusion, we were able to create a BlackJack game that met our personal requirements. We were able to take our text-based game and transform it into a nice, user-friendly graphical interface. We had more trouble than we would have liked linking the interface with the actual code, but were pleased with the results. EiffelStudio allowed us to create a wonderful game, and we made great use of design by contract.

FUTURE WORK

If time had permitted, we would have liked to implement the following:

DOUBLE AND SPLIT CAPABILITIES

We would have included the standard *Double* and *Split* options found in the traditional BlackJack game. *Double* doubles your bet and takes exactly 1 more card, while *Split* splits your cards into two separate hands that you *Hit* or *Stand* with appropriately. Doing this would have more accurately simulated the authentic BlackJack game.

INCREASED DEALER INTELLIGENCE

It would have been nice to implement more than just the *Soft 17* format for our dealer to follow. If time had permitted, we could have had an option where the user chooses a difficulty level and the dealer follows a strategy corresponding to that level. This would have improved gameplay and made the game more appealing.

MULTIPLAYER GUI SUPPORT/NETWORK SUPPORT

Our current GUI supports only 1-player games. The original text-based game had support for up to 8 players. Without us drastically changing the GUI's main window, it would be very difficult to allow for more than one player. If we did change the layout, multiplayer support would not have been too hard to achieve. Doing this also would have allowed us to implement network play.

REFERENCES

Professor Jesús A. Izaguirre — Consultation
http://docs.eiffel.com — Eiffel documentation
http://games.yahoo.com — Ideas for basic layout of game
http://www.casinoonnet.com — Basic background image
http://www.nd.edu — Notre Dame related graphics
http://www.pokerchips.com — Poker chips images

BIOGRAPHICAL INFORMATION

Mudit Agarwal is a Junior majoring in Computer Science. He is from Calcutta, India and he enjoys computers, movies, music, electronics, visiting places, and sports. He plans on returning to his hometown to work as a computer programmer for a reputable company.

Ryan Kennedy is a Junior majoring in Computer Science. He is from Niles, Michigan and he enjoys traveling, computers, movies, cars, and hanging out with his family and friends. He plans on earning an MBA after his Computer Science degree.

Mudit

Ryan