Formal Methods & Programming Coursework 2

Unit Code: CM50109

Programming with Oberon-2

Deck of Cards Demonstrator

Ryan Kelly

rmk22@bath.ac.uk

Table of Contents

1 Original Problem Statement

"You must develop a prototype software system that models standard decks of 52 playing cards and includes a demonstration of the usual types of actions required in typical card games. We have already identified three key actions that the system must perform: shuffling the deck, cutting the deck and dealing x cards to y players. Your solution must be modular."

2 Requirements Analysis

Overall Goal

The goal of this project is to develop a prototype software system that accurately models a deck of cards. The system should allow a user to perform a number of common actions needed during card games. Our end user has specified that the system need not represent an actual game; it should merely demonstrate that simple card game practices such as dealing and shuffling can be represented by a computer program. We anticipate that if our client is satisfied with the test program, he may take out a contract instructing us to create a program modelling a fully-functional card game.

2.1 Modular Solution

From the original problem statement, we have identified that our solution must be modular. We believe that the best course of action for this software development project is to implement the deck of cards as a true data abstraction, creating a library module to implement the deck of cards and then use a second module as the demonstration or test program.

2.2 Deck of Cards: Characteristics

1. *How many cards should be in a deck?*

There are 52 playing cards in a standard real-world deck. In a single deck there are four suits - respectively Clubs, Diamonds, Hearts and Spades. Within each suit are the following cards: 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King and an Ace. All of these properties should be accurately mirrored in the deck of cards and test program provided by our problem solution.

Decks of cards often include pairs of 'Joker' cards that are required in some card games. However, these will not be included in our system because the client has specified that the system should contain 52 cards, and incorporating the Jokers would bring this number up to 54. It is possible

that the client has overlooked the presence of the Jokers. Regardless, we feel that they are unnecessary for this test program.

2. *How many decks should the system model?*

Because some card games often use multiple decks of cards, there will be no restriction on the number of deck instances that the test program will be able to use. We find it unlikely that the user will require a large number of decks (as the end user is only looking to see that the program can perform basic functions such as dealing and shuffling) but we will include the ability to create multiple decks in order to meet the conditions of the original problem statement.

3. *Should the deck be subject to modification by the user?*

Our stance is that the user should not be able to arbitrarily add or remove cards from the overall deck at any time. For example, they should not be able to remove all of the Kings, or any particular card (e.g. the Ace of Spades). This is to prevent 'cheating' (see 'Security Requirements', pt. 16) and also because it has been decided that this kind of function is not required in the test system.

2.3 Deck of Cards: Required Operations

4. *What operations have already been identified as essential?*

The end user has already specified that the test program should be capable of shuffling the deck of cards, cutting the deck and dealing a specific number of cards out to a given number of players.

We will use a definition of card shuffling taken from an online poker glossary: shuffling is the 'act of mixing the cards before a hand, such that they are randomised'[1]. There are some issues with shuffling in professional card games, for example, false shuffling where the cards appear to have been shuffled but are actually set to a predetermined order. We do not anticipate this to be an issue in our system because the computer will be responsible for the shuffle, not the user. The user will only be able to specify whether or not a given deck should be shuffled.

We define cutting the deck as 'dividing the deck into two sections in such a manner as to change the order of the cards'[2]. We recommend that this action be offered to the user after the deck has been shuffled. The user should be able to specify where they would like the deck to be cut by giving the card number to the system (for example, at the 24th card).

Dealing is defined as 'giving each player cards, or to put cards on the board'[3]. We recommend that the process of dealing takes place after the user has been asked to shuffle and cut the deck. The

[1] Poker is a popular card game. Shuffle definition link: kofpoker.com/html/s.html
[2] Cut definition link: http://kofpoker.com/html/c.html
[3] Deal definition link: http://kofpoker.com/html/d.html

system will need to ask the user how many players are in the game and how many cards each player should receive.

5. What should happen to cards that are not dealt to players?

Any leftover cards should be placed into a 'pack'. Certain operations may then be carried out on this pack (for example, if a player discards a card from his or her hand, it should return to the bottom of this pack).

6. What other operations might the user need to perform during a card game?

Besides shuffling, cutting and dealing, most card games involve each player drawing cards from the pack and discarding cards from their hand, either to the bottom of the deck or into a central pool of cards, often known as the 'pile'. The user may also wish to see the results of each operation that the system performs, for example to examine the results of a shuffle.

As such, we recommend that the system allows the user to:

- Display a single card from their hand.
- Display each player's hand of cards.
- Display the entire deck (to provide evidence that it has been shuffled or cut).
- Throw away a card from the player's hand onto the 'pile'.
- Throw away a card from the player's hand to the bottom of the deck.
- Draw another card from the 'deck'.

These operations should only be available at the appropriate time, i.e. it shouldn't be possible to shuffle the deck again once the cards have been dealt. Further consideration should be given to these issues during the design process.

2.4 System Constraints

7. What constraints will need to be placed on the system?

We need to consider how many players are likely to be involved a card game. It is unlikely that the number of players would exceed the number of actual cards in a deck, so the absolute maximum number of players would be 52. We have decided to constrain the test program to dealing cards to a maximum number of 6 players. This is because our client has said that they will often play cards with '4 or 5 friends', thus we consider it unlikely that the user will wish to see actions taking place between more than 6 players in this test program.

The program could be expanded to cater to more than 6 players at a later date (see 'Maintenance Requirements' for a more in-depth discussion of this).

We have decided not to place a limit on the maximum number of cards each player can be dealt. Some games require small hands of 1-5 cards (e.g. Poker) whereas other games may require division of the entire deck (e.g. Snap). As such we feel that limiting the number of cards a player can receive is inappropriate for this test program. However, it should not be possible to deal more cards than there are in the current deck.

Speed is not a critical issue in this system but ideally the system should perform functions like dealing and cutting the deck in 3 seconds or less. It is fully expected that the system will be capable of meeting this level of performance as the functions will be relatively simple and consideration will be offered to memory usage during the design process.

2.5 Hardware & Software Constraints

8. Are there any hardware & software constraints?

Our client has asked that the program be coded using the Oberon-2 programming language via the use of a Windows PC. The finished system must be able to run on the machines in the MSc lab of the Computer Science department at Bath University. The Pow! Oberon-2 compiler should be used to code the program which will ensure compatibility with the software running on these machines. The use of a colour monitor will be necessary as the interface may involve some use of colour (see section on 'System Interface' for more information).

9. Are there any other constraints?

Three weeks are available for the design, implementation, testing, evaluation and delivery of the system. The finished system and its related documentation must be ready for delivery to the end user by 12:00pm on the 11th of December.

2.6 System Interface

10. How will the user operate the program?

If we were creating a fully functional card game we would recommend that the user be able to manipulate the deck of cards via a graphical user interface. However, due to the nature of this implementation (test program) and a lack of development time we recommend that the system is operated via a command line interface.

We anticipate that most of the commands the user will be required to enter will be of boolean format; for example, the question: "You have a deck of cards, do you wish to shuffle the deck? (Y/N)" requires a single Y or N character response. We recommend that the system be designed in this manner and we believe that a command line is more than capable of handling these types of input. Furthermore,

this interface should be more than sufficient for any other commands required from the user during the run-time of the test program.

11. How should the cards be represented on the user's screen?

The end user has not specified how the cards should be represented by the system interface. We have identified three methods of representing the cards within the test program: firstly, by representing each card graphically through the use of a small image representing each card; secondly through the use of codes such as 3Hearts, 5Clubs, etc., and finally by using a similar coding system but instead of a word, use a coloured symbol to represent each suit, for example 4 ♥, 5 ♠.

We consider the third method to be the most preferable because it incorporates the three types of information the observer needs to know about their card (the value, the suit and the suit colour) in a short and succinct manner. However, if our developer is unable to code this system then the code system of 3Hearts, 5 Clubs etc. will be used instead.

The system should provide suitably informative feedback to the user after each procedure has been carried out. This could come in the form of simple text-based feedback, e.g. "Shuffling cards" followed by "Your cards have been shuffled".

2.7 Data Input Required

12. When will the user issue commands to the system?

We have already stated the need to perform a shuffle operation, a cut and the dealing of cards to players. These actions are usually performed at the start of card games, so we recommend that the user be asked to perform these actions as they would in a normal card game (i.e. at the start) and they can then go on to perform other actions such as drawing cards from the pile or removing cards from their hand.

Other commands (such as removing a card from the user's hand and drawing a replacement card) should only be possible at the *appropriate time*, i.e. when the user is prompted by the system.

13. What kinds of input will be required?

As we have already suggested the use of a command line input, we recommend that input from the user should be in response to yes/no prompts from the system and should be of single character format, i.e. "Y" and "N". The system needs to be able to handle these kinds of input and should not accept erroneous commands. Some form of validation will need to occur each time the user enters a command.

When performing the Deal function, the system will need to accept at least two integers from the user, in order to determine the number of players in a game and the number of cards that should be dealt to each player. Thus we can explicitly state that the system needs to be able to handle character and numerical (integer) input.

2.8 Security Requirements

14. How many users will there be for this system?

We have only identified one type of user for the system - the end user. Because of the relatively straightforward nature of the test program, we do not feel that there is a need to consider issues such as password protection or multiple levels of security for administrators or "ordinary" users.

15. Are there any other security issues?

We have recognised that whilst the test program is running there may be issues relating to the possibility that a user is able to 'cheat', either by adding or removing cards from the deck or by altering a dealt hand of cards. However, we consider these security issues as relating to the design of the test program and to the integrity of the deck of cards itself, rather than the overall security of the system. As such, we believe that these issues are best addressed during the design process. However, at this early stage we recommend that the user should not be able to modify the *physical properties* of the deck of cards in any way, beyond performing legitimate functions such as shuffle, cut, etc.

2.9 Testing Requirements

16. What are the testing requirements for the system?

The end user has specified that the program should be tested using both black box and white box testing. Both of these should be done in a structured fashion after the test program has been written. In order to avoid problems related to syntax errors, we recommend each component in the system is tested informally during the coding process. For example, each time a new procedure is coded an attempt to compile and run the code should be performed.

The system is required to run on the computers in the MSc lab at Bath University. As the hardware of the computers in the lab can vary greatly, we recommend that the finished program be tested on at least 3 of these machines.

We are confident that these measures will ensure our system is suitable for delivery to the end user.

2.10 Documentation & Training Requirements

17. Is any documentation required?

The client has requested a fully commented print out of the finished program so that they can observe the Oberon-2 code. We recommend that this is done directly from the Oberon-2 compiler in

order to any avoid issues that may stem from the rearrangement of the code structure by a word processor program. The client has also requested copies of our requirements analysis and resulting specification, copies of our designs and design rationale, as well as a short evaluation, conclusion and critique of the overall system.

18. What are the training requirements for the system?

The end user is proficient in the use of Microsoft Windows and is able to run and compile an Oberon-2 program. However, our client should be provided with a short user documentation manual demonstrating how the system works and how it should be operated. We should also provide contact details to the customer so that they are able to contact us if they have problems with the system. If the end user has further difficulty learning how to operate the program, a short face-to-face demonstration should be provided at no extra charge.

2.11 Maintenance Requirements

19. Are there any maintenance requirements?

Our client has not given any indication of what maintenance (if any) they require for the system. Normally we would offer to maintain the system for a given length of time (possibly requesting a small fee for this service). However, it is felt that the finished demo program should not require any serious long-term maintenance as its purpose is simply to demonstrate to our client that a deck of cards can be accurately represented in Oberon-2.

We anticipate that this test program is the first step towards implementing a larger system which will accurately portray a working card game. If the end user is satisfied with the test program, we could offer to implement a working card game or offer to improve the test program by including additional functionality in the next version. This will need to be discussed between the developer and the user.

If the end user discovers any serious bugs within the test program that were not identified and corrected during testing, these will be dealt with by offering to rectify the test program at no extra charge.

2.12 Development Resources

20. What development resources are required?

This is an issue for the developer rather than the customer. The developer will need some knowledge of the programming language Oberon-2, a personal computer with the Pow! Oberon-2 compiler installed and enough hard disk space to save the finished object files. We recommend that the program be coded in the MSc lab at Bath University in order to ensure that the finished program runs on the available computers, but this is not essential. The finished program will need to be stored on a floppy disk. We recommend keeping backup copies on several different computers.

3 Requirements Specification

Each heading below refers to the corresponding section in our Requirements Analysis. The number after each statement indicates the question in the Requirements Analysis from which it was derived.

3.1 Modular Solution

- The problem solution must be modular.
- The deck of cards must be implemented as a data abstraction and must have its own library module.

3.2 Deck of Cards: Characteristics

- The system must accurately model the properties of a deck of cards. (1)
- The deck must contain 52 cards. (1)
- The deck must contain the appropriate cards (2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King and an Ace) distributed correctly amongst the four suits (Clubs, Diamonds, Hearts and Spades). (1)

- There should not be a limit on the number of decks a user can create. (2)

- The deck will not be modifiable. (3)

3.3 Deck of Cards: Required Operations

- The system must provide the user with at least one deck of cards. (4)
- Prior to dealing the cards, the system should provide the user with the opportunity to:
  - Shuffle the deck of cards (4)
  - Cut the deck of cards (4)
- The program should ask the user to indicate the location at which the deck should be cut. (4)
- The system must ask the user how many players they wish to deal cards to. (4)
- The system must ask how many cards each player should receive. (4)

- Leftover cards should form a ‘pack’ from which players can draw cards. (5)

- The system must allow the user to display single cards, a hand of cards or the entire deck. (6)
- The system must allow the user to throw away a card from their hand and draw a new card. (6)

- These operations should only be possible at the appropriate time. (6)

3.4 System Constraints

- The system must allow a maximum of 6 players per game. (7)
- The number of cards each player can receive must not be limited. (7)
- Operations must take no longer than 3 seconds. (7)

3.5 Hardware & Software Constraints

- The system must be implemented using the programming language Oberon-2. (8)
- The finished system must be capable of running on the computers in the MSc lab at Bath University. (8)

- The system and related documentation must be ready for delivery to the client by 12:00pm on the 11th of December. (9)

3.6 System Interface

- The system must use a command line interface. (10)
- The interface should be designed such that the actions are performed after the user is prompted to enter responses to questions. (10)

- The cards must be suitably represented in the test program. The use of the value/suit symbol system is the most preferable (4 ♥, 5 ♠, etc). (11)
- If the developer is not able to code the symbol system, the naming system of 4Hearts, 5Clubs, etc. must be used instead. (11)
- The system must provide informative feedback after each operation. (11)

3.7 Data Input Required

- The system must be able to handle simple user input. (12)
- The system should ask the user to shuffle and cut the cards after the deck has been created. (12)

- Input commands should be in response to program prompts. (13)
- Input commands should be single character (“y”, “n”, “s”, etc.) (13)
- The system must not accept erroneous commands and must validate all input. (13)

3.8 Security Requirements

- The program does not need any security measures to control who has access to the system. (14)

- The test program should not allow the user to arbitrarily change the physical properties of a given deck. (15)

3.9 Testing Requirements

- The system must undergo a suitable level of black box testing. (16)
- The system must undergo a suitable level of white box testing. (16)
- The program must be tested on at least 3 different available machines in the Computer Science MSc laboratory at Bath University. (16)

3.10 Documentation & Training Requirements

- A fully commented program printout (directly from the compiler program) must be provided to the client. (17)
- A document detailing requirements analysis and resulting specification, copies of the designs and design rationale, as well as a short evaluation, conclusion and critique must be provided to the client. (17)

- The client must be provided with a user documentation manual. (18)
- The manual should indicate how the system works and how it should be operated (and should include a full list of any commands they may need to learn in order to operate the system). (18)
- Company contact details should be provided to the end user. (18)
- If there are further difficulties, the client should be provided with a tutorial, free of charge. (18)

3.11 Maintenance Requirements

- Bug fixing should be offered to the client. (19)
- A maintenance service should be offered to the client. (19)

3.12 Development Resources

- The developer must have some knowledge of Oberon-2, access to the PoW! Compiler and enough hard disk space to save the finished program. (20)
- Backup copies of the system must be kept at a suitable location. (20)

4 Design

We now move on to the design of the deck of cards demonstration program. The creation of a well-planned and thoughtful design will allow us to achieve two goals. These are:

- To ensure that the system meets the requirements specification. We need to design the system as a modular solution.
- To give consideration to the process of implementing the deck and required procedures. This will save time during the development process. Forming conceptualisations of each module will aid our developer's understanding of the coding process and will help him to see what actions need to be carried out by the code to create the deck of cards. At the time of writing, our developer is a relative programming novice, so anything that will help the coding process is of benefit to this project.

4.1 Modular Decomposition

In order to implement a program that models a deck of cards and meets our specification, we have identified that the system will need three modules:

- A module implementing the deck of cards library.

The need for a module implementing the deck of cards is something of a given. Giving the deck its own library will also allow for reuse if our client decides to order a fully functional card game from us.

- A module implementing the demonstration program.

We believe that it makes good sense to keep the demonstration program separate from the cards library. The demonstration program will not actually alter the deck in any way; merely, it will call procedures that are responsible for such actions. Procedures that print single cards, hands or the entire deck will be included in the test program. This is because these functions have been identified as specific to our client (future clients may not require a deck that displays a hand).

- A module implementing a random number generator.

The concept of shuffling will require the use of a random number generator. At this point, we recommend that the procedure responsible for random number generation be housed in its own module. It can then be made available to client modules and procedures. Furthermore, creating an independent number generator means that the module can be utilised by future programs coded using Oberon-2.

We could also have chosen to implement a separate module implementing the concept of player ‘hands’. Our reasons for not doing this will be discussed later.

4.2 Data Representation

A deck is essentially a collection of 52 cards. Cards are similar in that every card has the properties of value and suit. However, the combination of these values is unique to each card; for example, the Ace of Hearts differs to the Two of Hearts in that whilst their values are different, their suits remain the same. Similarly, the Ace of Spades shares its value with the Ace of Hearts, but they have different suit values. As such, they can be seen as separate entities, so in our program we need a suitable method of storing the unique properties of each card in the deck.

The best way of doing this will be to utilise a ‘Card’ data type, with each card containing information about its suit and value. These properties are best represented by an integer value – so the Ace of Hearts might have a value of ‘1’ and a suit of ‘1’, with the Two of Hearts represented by a value of ‘2’ and a suit of ‘1’, etc. Of course, the cards will not be presented as integer values to our client; the cards will be represented using one of the two representation systems as specified in point 3.6 of the requirements specification.

A card will be represented as a ‘record’ in Oberon-2. A single card will contain the following fields:

Value: Integer.
Suit: Integer.
Next: Pointer to the next card’s location.

We have included a pointer in our data type because the deck of cards must be implemented as a dynamic data structure. This means that in order to access the location of each card in the system memory, there must be a pointer to that card. We will now discuss our deck and why it will be implemented as a dynamic structure.

4.3 Data Abstraction

In order to represent the deck of cards accurately, it must mirror the properties specified in the requirements specification (4 suits containing 13 cards each). Furthermore, the deck should be flexible enough such that it is possible to perform the following operations on the deck of cards: shuffle, cut, deal cards to players hands, allow players to throw away a card, allow players to draw a card. As mentioned previously, we aim to consign procedures that alter the deck itself to the library module, with the client program able to call these procedures as necessary.

In terms of achieving a deck that is open to innovative manipulations, we have a number of options.

4.3.1 Array of 'Cards'

In terms of creating a basic deck, our first option would be to create a fixed array structure of 52 cards. The benefit of this is that as the array is fixed, it becomes impossible to have an inappropriate number of cards in the deck, i.e. the deck cannot exceed 52 cards. Furthermore, accessing each card in the deck would be easy, as we could simply cycle down the array structure until the required card is found.

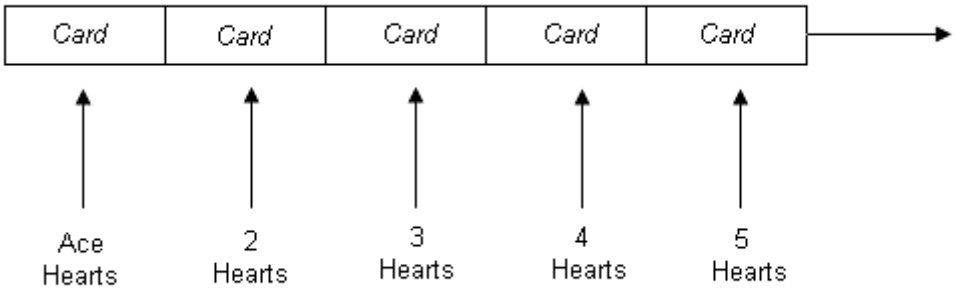

Figure 1: Showing how our card data type might fit into an array.

However, an array does not represent a dynamic data structure, so is inappropriate for this project as it will not fulfil the terms of our original project brief. Furthermore, moving items around the array represents a difficult task and this may impede our ability to program a shuffle function.

4.3.2 Queue Data Abstraction

Use of a queue would represent a more flexible solution to the problem of creating a deck of cards. A queue can be implemented as a dynamic data structure. Furthermore, the idea of removing cards from the top of the deck and inserting cards at the bottom mirrors the properties of a queue, whereby items at the front of the queue are removed first. However, it is unusual for items in a queue to be moved around at random; queues represent the idea of an ordered list of waiting items.

4.3.3 List Data Abstraction

We consider the use of a list as the most appropriate method of representing the deck in Oberon-2. The list will not only allow us to remove and add items but conceptually represents a more suitable representation than a queue. This is because lists can often be subject to change or be rearranged at will. However, there are two different types of list – single linked lists and ordered lists. We have

chosen to use a single linked list because, although our cards do have an initial 'order' according to their suit and value, this is subject to change from shuffling and cutting.

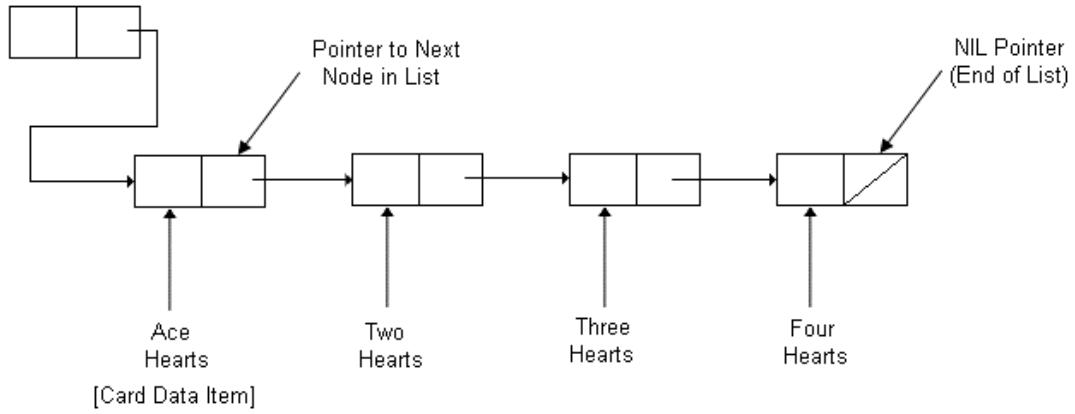

Figure 2: Showing how the nodes of a linked list contain data and point to the location of the next node in system memory.

Each node in our list will contain a single card. The nodes can then be switched individually during our shuffle. After the results of a shuffle, the list might look something like this:

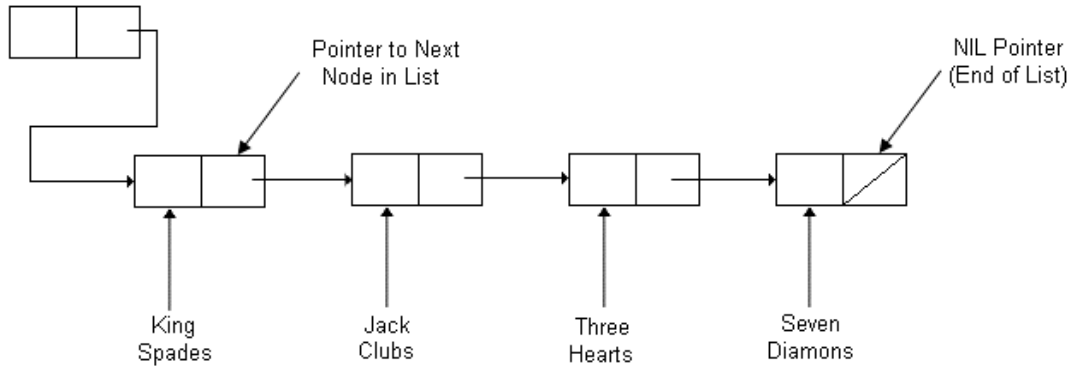

Figure 3: Showing how the cards have changed positions without damaging the integrity of the list structure.

The image demonstrates that whilst the structure of the list itself does not need to be altered, nodes can be switched at will. A further benefit of the list is that when we deal cards to a hand, the list can shrink. A fixed structure like an array will continue to use system memory even if some of its nodes are empty. So in terms of memory usage, a list is far more efficient. We do not see memory usage as a problem during this project, but it does become a concern during large programming tasks.

The use of a list will also make cutting the deck very simple. We will need to determine the location of the cut, cycle down the list and alter the pointers as necessary. Performing a cut on array would be much more difficult.

4.3.4 Other Abstractions

We could have also chosen to implement the deck of cards as a stack. However, we felt that this was inappropriate because stacks operate on a ‘first in, last out’ basis, and in our deck we may need access to any given card in the data structure.

4.3.5 Modelling Hands

In order to carry out a deal procedure, we need to have a concept of ‘hands’. When the user performs a deal, they will be asked how many cards should be dealt to the ‘hand’ of each player. Hands could be implemented in a separate module, with each ‘hand’ formed from a small linked list, in turn receiving cards from our ‘deck’ list. These ‘hands’ could then form a list – effectively creating a ‘list of lists’. However, our requirements specification has specified that the system will not need to model more than 6 hands at one time. As such, we plan to store our hands using a fixed sized array – essentially an array of 6 hands maximum. This is because at this stage we would like to aim for simplicity in our program so that the coding process remains as simple as possible for our developer. Procedures implementing and making use of hands will be designed with the array structure in mind.

4.3.6 Concept of a Pile

In our original specification, we stated that players should be able to throw cards onto a pile. At this stage, we would like to revoke this requirement because we do not foresee our developer having the time to implement the pile. Furthermore, we would like to focus on coding the basic operations required to meet the original problem statement. We feel that the loss of the pile will not impact the overall goal of the system, which is to demonstrate to our client that a deck of cards can be modelled by a computer program.

4.4 Formal Specification of Modules

We now move on to the description of the modules to be implemented in the program, along with their related procedures. We have also outlined the algorithm for each procedure – the algorithm represents the steps the program should take to achieve the goal of each procedure.

4.5 Module CardDeck;

This module will contain all procedures related to the deck of cards.

Key properties: Library module, implements deck of cards and related procedures, makes procedures available to the client module.

```
DEFINITION OF MODULE CardDeck;
TYPE
        CardPtr = POINTER TO Card;
        DeckPtr* = CardPtr;
        Hand* = CardPtr;
        Card = RECORD;
                value : INTEGER;
                suit : INTEGER;
                next : DeckPtr;

PROCEDURE NewCard(integer; integer; DeckPtr;) : DeckPtr;
PROCEDURE CreateDeck*(): DeckPtr;
PROCEDURE CutDeck*(DeckPtr);
PROCEDURE Shuffle*(DeckPtr);
PROCEDURE CreateHand*(): Hand;
PROCEDURE Deal*(DeckPtr; ARRAY OF Hand);
PROCEDURE ThrowCard* (integer; DeckPtr; ARRAY OF Hand);
PROCEDURE DrawCard* (integer; DeckPtr; ARRAY OF Hand)
```

Procedures & Related Algorithms

4.5.1 **NewCard**: (INTEGER; INTEGER; DeckPtr) → DeckPtr;

This procedure implements a single card.

Pre Condition: None.

Post Condition: A new card is returned.

- **Related Algorithm**

```
Begin
    Create a pointer to a new card.
    Assign a value to each field of the card.
    RETURN the new card.
End;
```

- **Explanation of Algorithm**

This algorithm is designed to be called by CreateDeck. It will create a single card, taking integer values for the suit and value fields from CreateDeck. This card is then returned.

4.5.2 **CreateDeck***: DeckPtr → DeckPtr;

This function implements a deck of cards by creating cards and assigning each a value. The deck is then made available to the user. DeckPtr points to the top card of the deck.

Pre Condition: None

Post Condition: A new deck is returned.

- **Related Algorithm**

```
Begin
    Call NewCard 52 times.
    Each time NewCard is called, assign each card with a suit and value.
    Make pointer to the previous card.
    Save the pointer to the top of the deck.
    Return the deck.
End;
```

- **Explanation of Algorithm**

This algorithm will create a deck by creating 52 cards and assigning appropriate values to the fields of each card. The pointer to the top of the deck is then returned.

4.5.3 CutDeck*; (DeckPtr) → ;

This procedure takes a deck of cards and cuts it at a specified location. The top 'half' created by this cut is placed below the lower half.

Pre Condition: A deck of cards has been created.

Post Condition: The deck is cut at a specified location and returned.

- **Related Algorithm**

```
Begin
    Create three temporary pointers to the top of the deck.
    Determine the location of the cut.
            Direct a pointer to the bottom of the deck.
            Direct a second pointer to the location of the cut.
            Make the card after the cut location become the top of the deck.
            Make the old bottom of the deck point to the old top card.
            Make the location of the cut the new deck bottom.
    Save the changes to the deck.
End;
```

- **Explanation of Algorithm**

This algorithm will cut the deck at a location specified by the user.

4.5.4 Shuffle* (DeckPtr) → ;

This function takes the deck of cards and randomises their order, effectively performing a shuffle operation on the deck.

Pre Condition: A deck has been created.

Post Condition: The order of cards within the deck is rearranged and returned.

- **Related Algorithm**

```
Begin
    Call a random number to determine the location of a random card.
    Create a pointer to this card.
    Call a second random number to determine the location of a second card.
    Create a pointer to this card.
    Swap the cards.
    REPEAT this process a suitable number of times.
    Save the changes to the deck.
End;
```

- **Explanation of Algorithm**

This procedure will shuffle the deck by switching the positions of two random nodes in our list. This is done multiple times to create the effect of a shuffle.

4.5.5 CreateHand*; → Hand;

This procedure creates a new hand in the form of a single NIL pointer, which can then be added to by Deal.

Pre Condition: Deck has been made available.
Post Condition: An empty hand, containing a single card is returned.

- **Related Algorithm**

```
Begin
    Create an empty hand in the form of a NIL pointer.
    RETURN this hand.
End;
```

- **Explanation of Algorithm**

This algorithm will create an empty hand. A hand will consist of a CardPtr, which is set to NIL.

4.5.6 Deal*; (DeckPtr; ARRAY OF Hand)

With the prerequisite that the deck has been shuffled and cut, this procedure deals a specified number of cards to a specified number of players.

Pre Condition: Deck has been created.
Post Condition: N number of cards are dealt to M players.

- **Related Algorithm**

```
Begin
    Determine the number of players (max 6).
    Determine the number of cards they should each receive.
    For each player:
            Create a hand.
            Deal the appropriate number of cards into their hand.
            Store the hand in the array
End;
```

- **Explanation of Algorithm**

This algorithm determines the players in a game and how many cards each should receive. It will then create the appropriate number of hands and assign cards to each hand as necessary.

4.5.7 **ThrowCard**; (INTEGER; DeckPtr; ARRAY OF Hand) → Hand;

This procedure takes a specified card from the player's hand and returns it to the bottom of a deck.

Pre Condition: Cards have been dealt into hands.
Post Condition: A card is removed from a hand and returned to the bottom of the deck.

- **Related Algorithm**

```
Begin
    Determine the hand the player wants to access.
    Determine the card the player wants to throw.
    Remove the card from the hand.
    Make the nil pointer at the bottom of the deck point to the thrown card.
End;
```

- **Explanation of Algorithm**

This algorithm will throw a card from a user's hand to the bottom of the pile.

4.5.8 **DrawCard**; (INTEGER; DeckPtr; ARRAY OF Hand) → Hand;

This procedure draws a fresh card from the top of the deck and adds it to the player's hand, provided they have a free slot.

Pre Condition: The user has a hand of cards with a free space.
Post Condition: A card is drawn and added to their hand.

- **Related Algorithm**

```
Begin
    Determine the hand the player wants to access.
    IF there is a free slot in the hand THEN
            Draw a card from the top of the deck and add it to the hand.
            Make the top of the deck become the card below the drawn card.
    Save the altered hand.
End;
```

- **Explanation of Algorithm**

This algorithm will draw a card from the deck and add it to a player's hand. The card below the drawn card then becomes the top of the deck.

4.6 Module Demo;

This module will act as the interface between the user (our client) and the deck of cards abstraction. It will consist of a ProgMain procedure which creates the command line interface that the operator uses to perform actions on the deck. Procedures will be called from the library module by the test program as necessary, according to user input.

Key properties: Demonstration program, creates interface between client and program, calls procedures from library modules.

DEFINITION OF MODULE Demo;

PROCEDURE PrintDeck (CardDeck.DeckPtr)
PROCEDURE PrintHand (INTEGER; ARRAY OF CardDeck.Hand);
PROCEDURE ProgMain* (CardDeck.DeckPtr);

Procedures & Related Algorithms

4.6.1 PrintDeck; (deck : CardDeck.DeckPtr) →;

This procedure prints the deck in its current form into the program window.

Pre Condition: A deck has been created and the user has asked to print.
Post Condition: The deck, in its current form, is printed to the program window.

- **Related Algorithm**

```
Begin
   WHILE the current card # NIL,
   Print the value of the card.
          Represent Aces, Jacks, Queens and Kings appropriately.
   Print the suit of the card.
          Represent the suits of Clubs, Diamonds, Hearts and Spades
          appropriately.
   Move to the next card.
   Until the next pointer is NIL.
End;
```

- **Explanation of Algorithm**

The algorithm will cycle down the deck, printing each card in turn. Appropriate phrases are assigned to face cards and the suit of every card.

4.6.2 **PrintHand**; (INTEGER; ARRAY OF CardDeck.Hand) →;

This procedure prints the contents of a hand into the program window.

Pre Condition: Hands have been created and filled with cards.

Post Condition: A specified hand is printed to the program window.

- **Related Algorithm**

```
Begin
    Determine the hand the player wants to access.
    WHILE the current card in the hand # NIL,
    Print the value of the card.
           Represent Aces, Jacks, Queens and Kings appropriately.
    Print the suit of the card.
           Represent the suits of Clubs, Diamonds, Hearts and Spades
           appropriately.
    Move to the next card.
    Until the next pointer is NIL.
End;
```

- **Explanation of Algorithm**

This algorithm is very similar to our print deck procedure. However, this procedure merely cycles through a single hand, printing each card in turn. Appropriate phrases are assigned to face cards and the suit of every card.

4.6.3 **ProgMain*;** (deck : CardDeck.DeckPtr) →;

This procedure prints creates the interface between the user and the deck of cards and calls the required procedures.

Pre Condition: None.

Post Condition: The program runs as expected.

- **Related Algorithm**

```
Begin
    Call the CreateDeck procedure.
    Ask user if they would like to view the deck.
           IF yes: call PrintDeck;
           ELSE
    Ask the user if they would like to shuffle the deck.
           IF yes: call Shuffle;
           ELSE
```

```
    Ask user if they would like to view the deck.
           IF yes: call PrintDeck;
           ELSE
    Ask the user if they would like to cut the deck.
           IF yes: call CutDeck;
    Ask user if they would like to view the deck.
           IF yes: call PrintDeck;
           ELSE
    Inform user they can deal. Call Deal;
    Ask user if they would like to view a hand.
    IF yes: call PrintHand;
    ELSE
End;
```

- **Explanation of Algorithm**

Our ProgMain forms the basis of our demonstration program. Here, procedures will be called in turn according to responses given by the user.

4.7 Module Random;

We need a module to generate random numbers for our shuffle function. The code in this module will be adapted from a random number generator written by Dr Claire Willis, as featured on page 144 of the lecture notes for the MSc: ACS Formal Methods and Programming unit (CM50109) at Bath University. The random number generator will be adapted such that it uses the system clock's current time as a starting point for the generation of a seed. This is because the system clock is the only thing we have access to that is likely to have a different numerical value every time it is accessed.

DEFINITION OF MODULE Random;

PROCEDURE Rand;

4.7.1 Rand*(): INTEGER;

This procedure will generate random number.

```
Pre Condition: None.
```

- **Related Algorithm**

```
BEGIN
    Access the current time, according to the system clock.
    Use the current second, minute and hour values at the time of access to produce
    a number, which is assigned to a variable.
    Use this number as a starting point to find a random number.
END;
```

4.8 Interface Design

We aim to keep our interface design as simple as possible. Our requirements specification stated that the user should input commands based on responses to a selection of yes/no questions. We believe that coding our demonstration program such that it asks the user the following questions should be sufficient to demonstrate our deck.

- Would you like view the deck?
- Would you like to shuffle the deck?
- Would you like to cut the deck?
- Would you like to deal out cards?
- Cards have been dealt, would you like to view the individual hands?
- Which hand would you like to view?
- Would you like to terminate the program?

Each of these questions will be followed by '(Y/N)' which should indicate to the user what kind of response they should give the program. After each response has been entered, appropriate feedback will be given to the user regarding the actions the program has taken. We recognise that the user may wish to view the deck after each operation, so they will be asked if they would like to view the deck after shuffling, cutting and dealing.

We feel that these questions should be sufficient to facilitate our user's navigation through the demonstration program.

5 Implementation

The code has now been implemented in Oberon-2 and we now have a working deck of cards. We will now move on to testing the system. During the implementation, we did not have time to implement some of the procedures we initially specified – reasons for this will be discussed in our evaluation.

It is also worth noting that we did not implement our shuffle as we originally intended. Originally, we planned to swap actual nodes within our list; what we actually found during coding was that it was much similar to simply exchange the value and suit properties between the two cards. This gives the impression that the deck is shuffled without actually reordering the list.

Please see Appendix C for a full print out of our code listings.

6 Testing

During the implementation of the code, each procedure was tested after it was written to ensure it operated as expected. To the best of our knowledge, the system operates as it should do. However, it is highly unlikely that we have created a flawless program. There are likely to be numerous bugs in various parts of the system. As stated in our requirements specification, we will test the system using some simple black box testing. We will also test the main pathways through the system using some simple white box testing. All test screenshots can be found in Appendix B.

6.1 Black Box Testing – Demonstration Program

We will now run the system through some simple black box testing. Each input from the user will be tested and verified to ensure that the program operates correctly and does not accept erroneous input. We deem the best way to do this as testing each user input one by one.

6.1.1 Black Box Test Plans

The user is asked to enter responses to yes/no prompts. We will test each of these inputs one by one to ensure that they do not accept invalid commands. Since these inputs are all of the same value, they can all be tested using the same black box test plan.

User Inputs

Input Characteristics	Valid Class of Data	Invalid Class of Data
Single alphabetic character.	“Y”, “y”, “N”, “n”	Any letters, numbers or symbols other than these four characters.

We have designed appropriate tests for each yes/no prompt in the demonstration and can now carry these out. Results are indicated below. Each yes/no prompt was tested using these criteria and alterations were made to the code as necessary.

Characteristic Under Test	Test	Expected Result	Actual Result
Response is positive: “Y”	“Y”	Accepted	All accepted. System carries out appropriate steps.
Response is positive: “y”	“y”	Accepted	All accepted. Screenshot B.1
Response is negative: “N”	“N”	Accepted	All accepted. Screenshot B.2
Response is negative: “n”	“n”	Accepted	All bar one were accepted; program was rectified as necessary. Retesting validated the alteration.
Response is an inappropriate character.	“1”, “ ”, “!” “P” “[}”	Rejected. Feedback given.	All rejected. Appropriate feedback given. Screenshot B.3
Response is a string of multiple characters.	“adhadh” “121pow” “***…**”	Rejected. Feedback given.	All rejected. Issue identified if the string contains a valid character, i.e. “y”, “Y”, etc. Attempted to rectify but required major alteration of the code. See evaluation for discussion of this. Screenshot B.4

See Appendix B for screenshots of testing process.

6.2 White Box Testing – Demonstration Program

Because the design of our system interface is quite linear, it should be possible to test the system using some simple white box testing. We will do this by testing different pathways through the system to ensure that they are all successful. It is difficult to demonstrate this process to the reader; nevertheless screenshots are included in Appendix B (Screenshots B.5 & B.6)

6.3 Black Box Testing – Card Library

We will now do some simple black box testing on the inputs required by our procedures in the CardDeck library module.

Characteristic Under Test	Test	Expected Result	Actual Result
Shuffle	Call shuffle multiple times and observe that results are different	Cards are shuffled in different orders.	Success. Screenshot B.7 & B.8
Cut Function – correct input	Numeric characters between 1-52	Deck is cut correctly	Success
Cut Function – input out of bounds	Numeric value out of bounds, i.e. >52	User is told that the number is inappropriate	Program crashes – catch added to code so that program will not accept a cut location higher than 52.
Cut Function – cut correctly.	Cut a shuffled deck and an unshuffled deck.	Cut occurs regardless of deck status.	Success.
Deal	Deal cards to 1-6 players	Cards dealt.	Success.
View Hands	User asks to view a hand	Correct hand is displayed.	Success.

6.3.1 Black Box Test Results

We have uncovered a number of problems during black box testing that were rectified in the code accordingly. Our cut function would crash if given a cut location that was greater than the deck itself, so a catch was added to the code to prevent this from happening. We have also identified that it is possible to attempt to view a hand that has not been dealt any cards; this should not be possible and we are currently working on a fix, but are unlikely to arrive at a solution before the deadline for this project. The issue does not cause the program to malfunction or crash, so we are aiming to finish this document and then return to the code and rectify the problem at a later date.

6.4 White Box Testing – Card Library

Putting the procedures in our CardDeck module through full white box testing is beyond the scope of this project. Testing every possible permutation of the shuffle function would take far too long and is unnecessary. We are confident that the procedures in the library module operate as expected.

6.5 Other Testing Results

There were a number of issues that occurred during the actual coding of the system. As good practice, we would like to explain some of the problems encountered and describe how they were solved.

When we initially implemented and tested our shuffle function, the program would repeatedly crash. The crashing was caused somewhere between the code and the random number generator – eventually a fix was discovered for the problem. Furthermore, our shuffle function worked correctly bar one issue – a specific card was never shuffled. This was discovered to be a problem with the loop responsible for the shuffle – for some reason the king of clubs was never shuffled but altering the loop fixed the issue.

Should the user pass the program a character instead of an integer when specifying a cut location, this sometimes causes the program to crash or infinitely loop. We recognise that it should not be possible to accept anything other than an integer into this field, but have not been able to code anything that acts as a preventative measure to the user giving incorrect input. We see this as a fault with the compiler, not with our program, as the compiler brings up a small error window whenever the user enters a character into an integer field.

7 Discussion

7.1 Evaluation

The card demonstration program and its related library modules have now been successfully implemented and tested. A command line interface has been included which leads the user through the system. The system carries out functions as our client expects them to be carried out during a real game of cards: a deck of cards is presented to the user and can then be shuffled, cut and dealt. The deck can also be viewed after any of these actions have been carried out.

We have resolved a number of issues discovered during the testing process and tightened up the functionality of the demonstration program. Where possible, we have also ensured that each input taken by the program is of the correct type. However, the use of Y/N responses presented problems with the input of large strings of random characters. If a string contained a valid response, i.e. “Y” or “N”, the program would accept this input then continue to process the invalid data. However, it was felt that this would not a major issue because the user is made aware that they are only required to enter one character, so the only time a random string would occur is if they were purposely trying to break the demonstration program.

In terms of meeting the requirements laid out in our initial analysis and specification, we feel that the program does a good job. The program fulfils its goal of demonstrating to our client that it is possible for a computer program to accurately replicate of deck of cards. We are pleased with the command line interface; whilst it is quite bland, it helps to navigate the user through the program and allows the user to carry out actions in a similar order they would expect to experience during a real card game.

That said, the finished is not perfect system and we recognise that a number of requirements from our initial specification have not been met. These requirements were centred upon several functionalities that we originally hoped to include but are now lacking. The demonstration does not have the ability to allow players to throw away cards from their hands back into the deck or onto a pile, nor does it allow them to draw cards - unfortunately these procedures could not be included due to time constraints. We feel that stating these additional functions in our requirements was something of a mistake, as we severely underestimated the difficulty of implementing the code for a novice programmer throughout the entirety of this project. This project was a significant challenge for our coder, who is a relative novice, and the functionality is missing due to a combination of time constraints and gaps in conceptual knowledge.

We also specified that the cards may be represented using a notation system whereby suits are represented by their respective coloured symbol, e.g. 4 ♥, 5 ♠. We believe that it would theoretically be possible to implement this system using Oberon-2, but again due to time constraints we were unable to include this. However, the naming system used in our demonstration program is more than satisfactory to meet the original project brief.

The overall challenge of designing, implementing and testing the system was a significant one for our developer. At times the process of coding was difficult and arduous, particularly with so little experience regarding the use of pointers. The most challenging task was the conversion of concepts related to shuffling, dealing and cutting into Oberon-2. Often, our developer knew what he wanted to do but struggled to actually implement it due to a lack of procedural programming knowledge. Thus, whilst the system is not perfect, the finalised implementation of the deck of cards is extremely pleasing when we consider that our developer has a background in psychology, not software development. We feel that spending care during our initial designs and reading around subject material was particularly beneficial during the coding of the system.

If we were to repeat the experience of designing the deck of cards, a number of things would be subject to change:

Requirements Analysis & Specification

- More consideration would have been given to deciding which operations to perform on the deck. In retrospect, it would have been better to state that we would fulfil the basic goals of the assignment brief before attempting to code extra operations.

Design

- Some of our procedures could have been more carefully designed. For example, our cut procedure involves asking the user to specify the cut location. It may have been more appropriate to ask for this in the ProgMain and then pass it to the cut procedure. The fact that some inputs were collected from the user in procedures rather than in the test program made it difficult to test individual components in isolation.

Demonstration Program & Implementation

- The overall pathway through the demonstration program is quite linear. Implementing a menu-based demonstration program, where the user could shuffle, cut and deal at will, could have been more interesting for both the programmer and the end user of the program.
- The demonstration program does not offer the user the opportunity to shuffle the deck multiple times. We recognise that this was an oversight during the design of the test program; if we were to re-implement the system, it would be possible to repeatedly shuffle the deck. This would be more representative of shuffling in the real world, as players often repeatedly shuffle cards until they are satisfied with the results.
- Our shuffle function was implemented slightly differently to how we had originally intended in that instead of swapping whole card records, only the value and suits are swapped between the

cards. This is a relatively simple way of reassigning cards around the deck and is still a valid method because card values and suits do not need to be tied to a specific node in our list.

- The use of Y/N responses presented problems with the input of large strings of random characters. If the user enters a random string of characters, they are not accepted, but if one of these characters is a valid response, i.e. a “Y” or an “N” then it is accepted and the program will attempt to process the other characters in the string. This is a problem when a yes/no response is immediately followed by an integer input.
- The time planning during the project could have been better. Further consideration would have been given to how the project could have been completed around other assignments.

Other

- Oberon-2 is a nice language but the Pow! Compiler is not the greatest program ever – our developer experienced difficulties with saving files and preparing the finished program for submission by floppy disk. Furthermore if the user presses the ‘End’ button in the compiler window, the program will infinitely loop forever. Also, it is difficult to validate integer input; if the user enters a character into an integer field, the program pops up with an error message.
- With more time, the additional functionalities we missed would have been implemented!

7.2 Conclusion

After initial difficulties with the actual process of programming during this project, we can only be pleased that a finished document and program are now ready to be delivered to the client. Whilst the program includes the functionality necessary to solve the initial problem statement, it was unfortunate that additional functionalities were not included. However the finished program is of good quality and our novice programmer is satisfied with his work. This assignment has been an uphill struggle at times, but has presented us with a significant learning opportunity. We look forward to receiving further opportunities in the future.

Appendix A

User Documentation

Oberon-2 Deck of Cards Demonstrator
User Manual

Introduction

Congratulations on acquiring the Oberon-2 Deck of Cards Demonstrator!

The Deck of Cards Demonstrator has been designed to accurately replicate a deck of cards and allows you to perform some simple operations on the deck. You can shuffle the cards, cut the deck and deal cards out to a maximum of 6 players. You can then observe the contents of each player's hand of cards.

Using the Program

To start the program, double click on the icon entitled 'demo' on the Deck of Cards diskette. You will then be able to access your deck of cards.

In order to manipulate the deck of cards, you will be required to answer questions, for example 'do you want to shuffle the deck?' Answers should be given as either Y or N responses so that the program is able to understand your responses.

The system will occasionally ask you for numerical input. **When the system asks you for a number**, **you must enter a valid whole number**. **If you do not do this, the program will crash and you risk damaging your system**.

Problems or Bugs in the Program?

We have done all that we can to ensure the smooth operation of the system. The system has been tested extensively but we cannot guarantee that bugs do not exist in the program.

If you suffer from a recurring bug, problems with crashes or encounter any further problems with the system, please do not hesitate to contact our customer service team at: rmk22@bath.ac.uk

We also offer tutorials at no extra cost.

Appendix B

Test Results & Screenshots

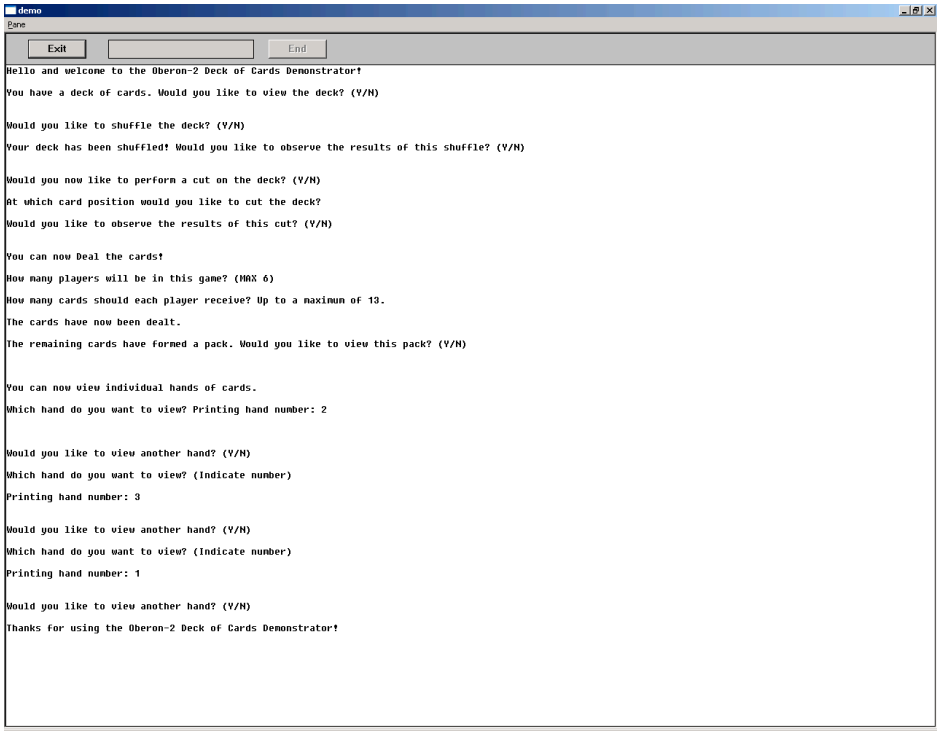

Figure B.1 – Screenshot showing appropriate acceptance when answering all prompts with affirmative answers.

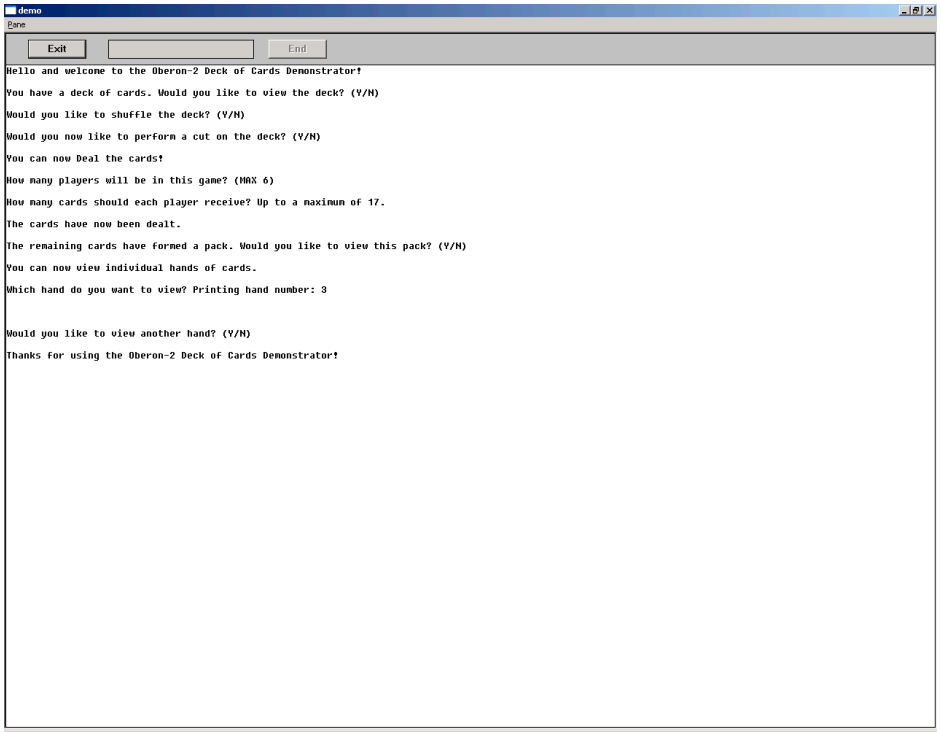

Figure B.2 – Screenshot showing appropriate acceptance of negative responses.

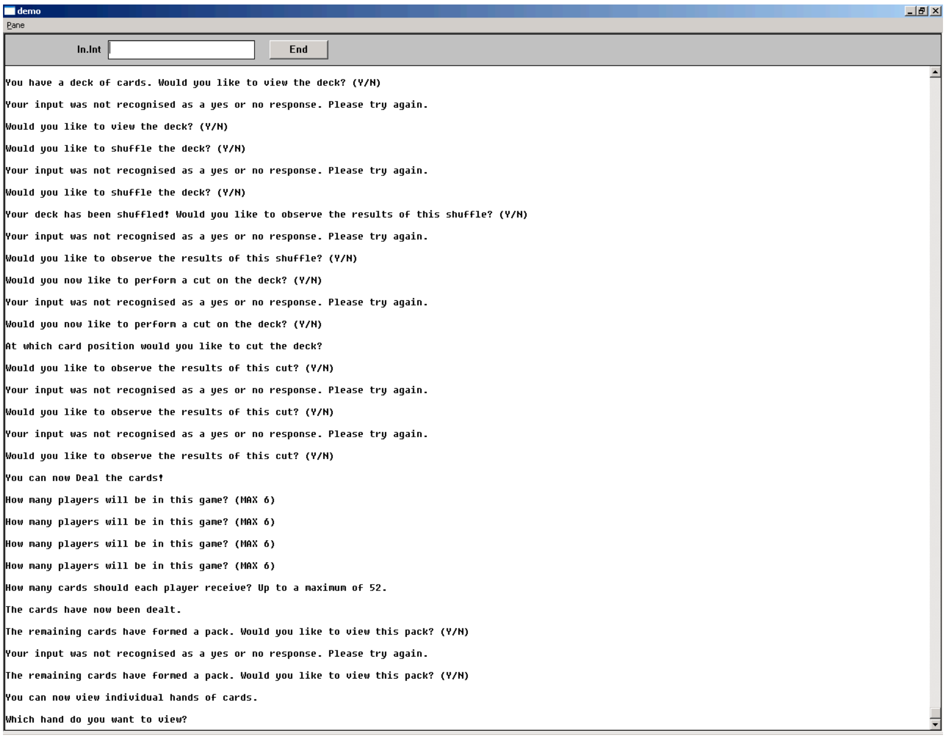

Figure B.3 – Showing how invalid inputs are not accepted by the program.

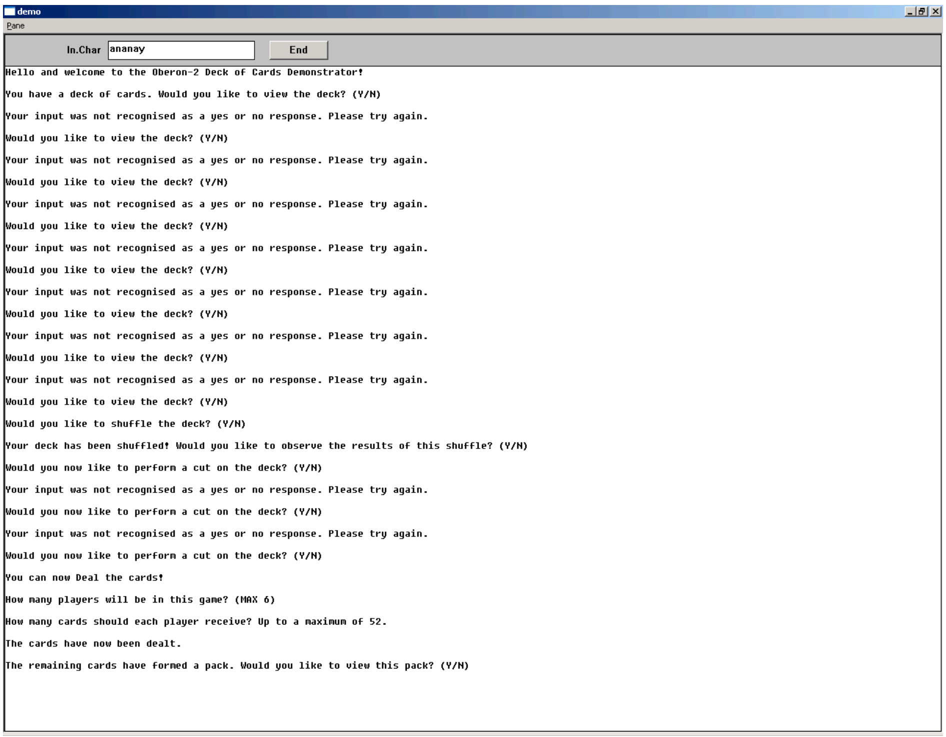

Figure B.4 – Showing how random strings of characters are not accepted by the program. There is a problem with this – if the string contains a valid character, this character is accepted by the program continues to invalidate anything else that comes after it.

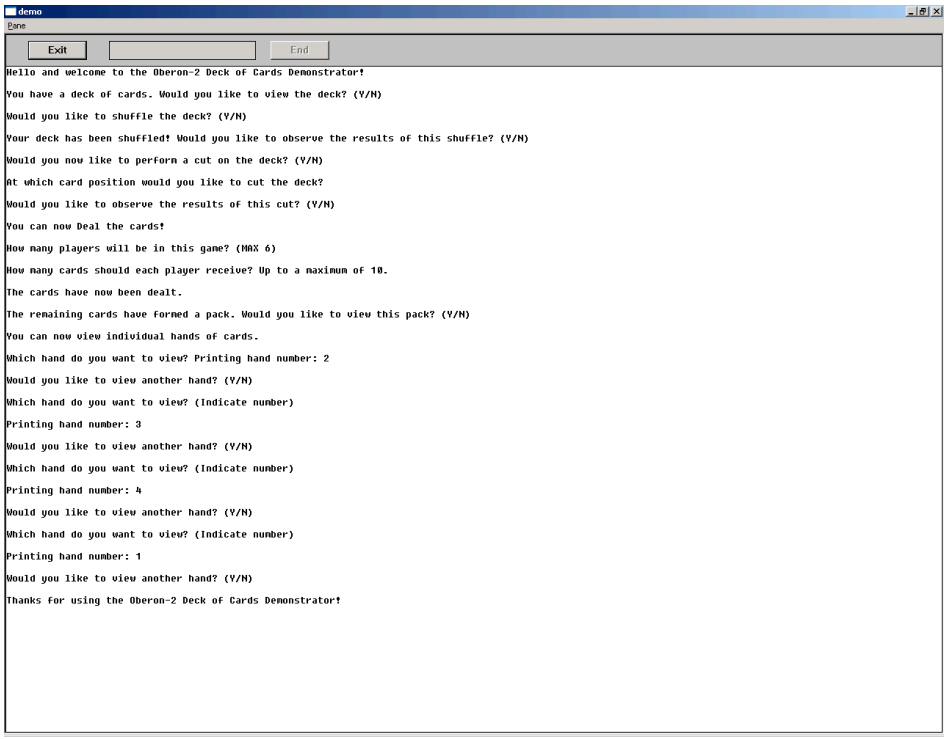

Figure B.5 - Showing white box testing for our Demonstration program.

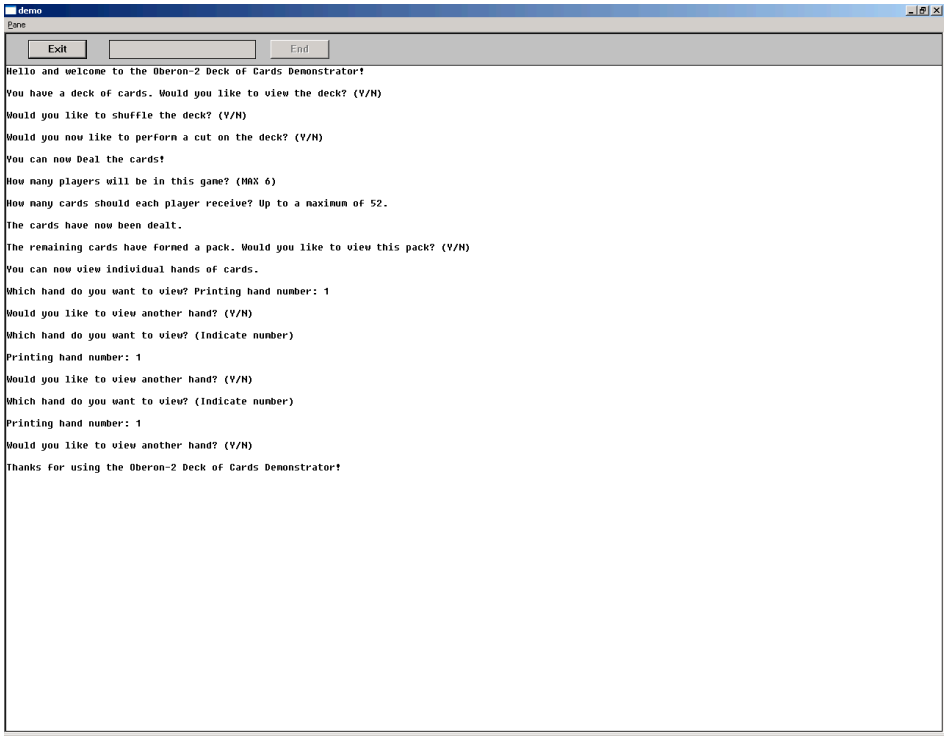

Figure B.6 – Showing another permutation of white box testing for our Demonstration program.

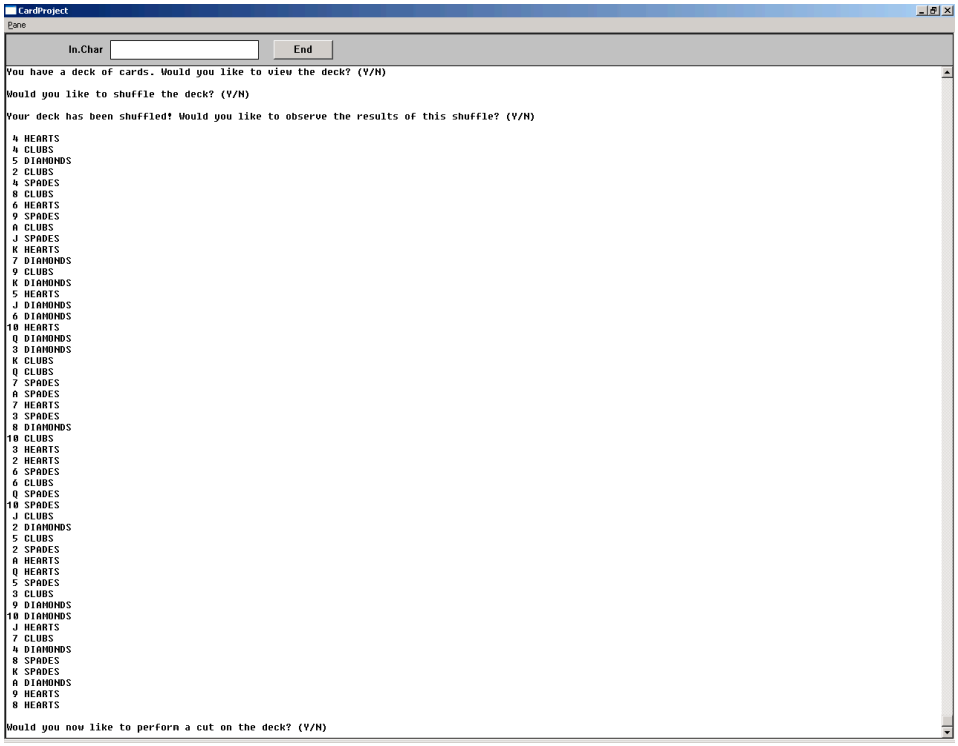

Figure B.7 – showing results of a shuffle

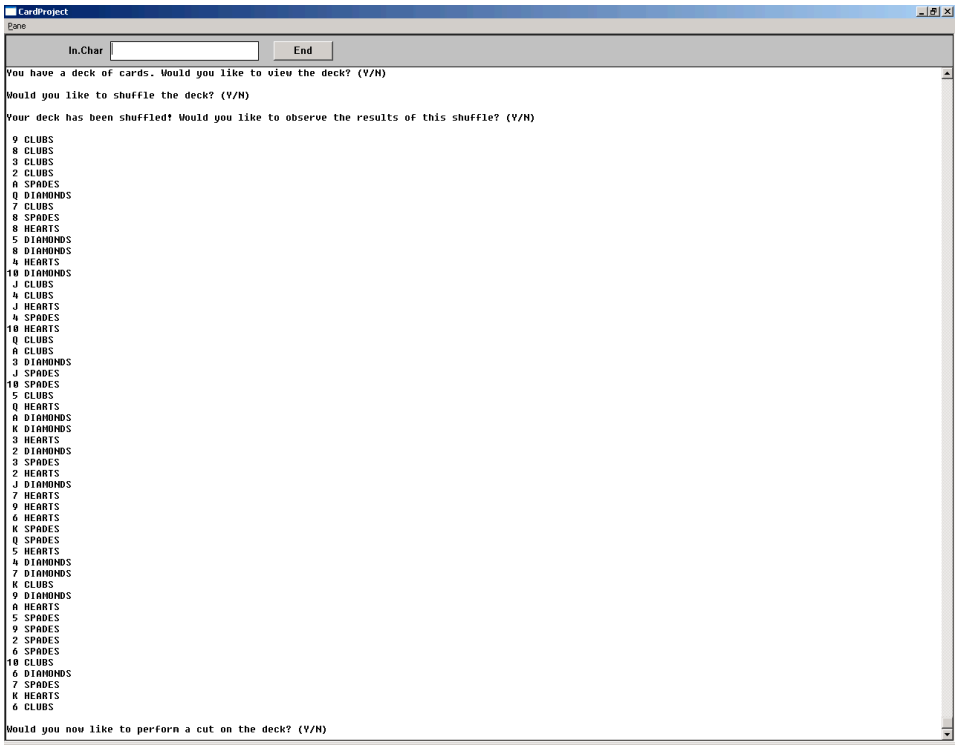

Figure B.8 Showing how the results of a second shuffle differ.

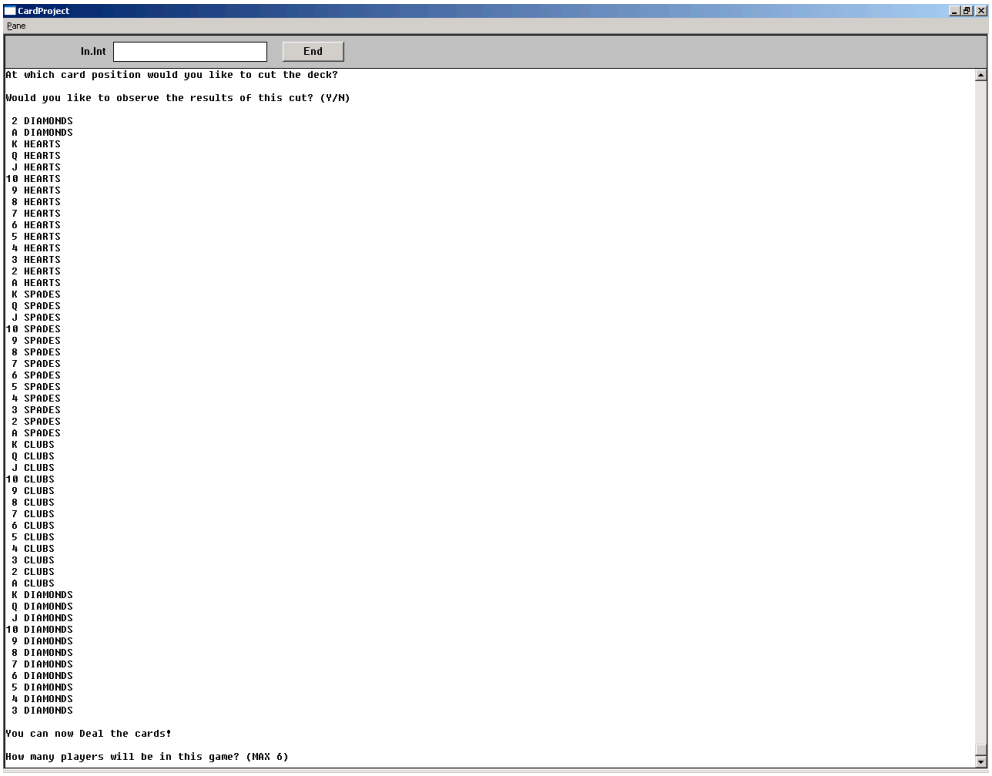

Figure B.9 Results of a cut.

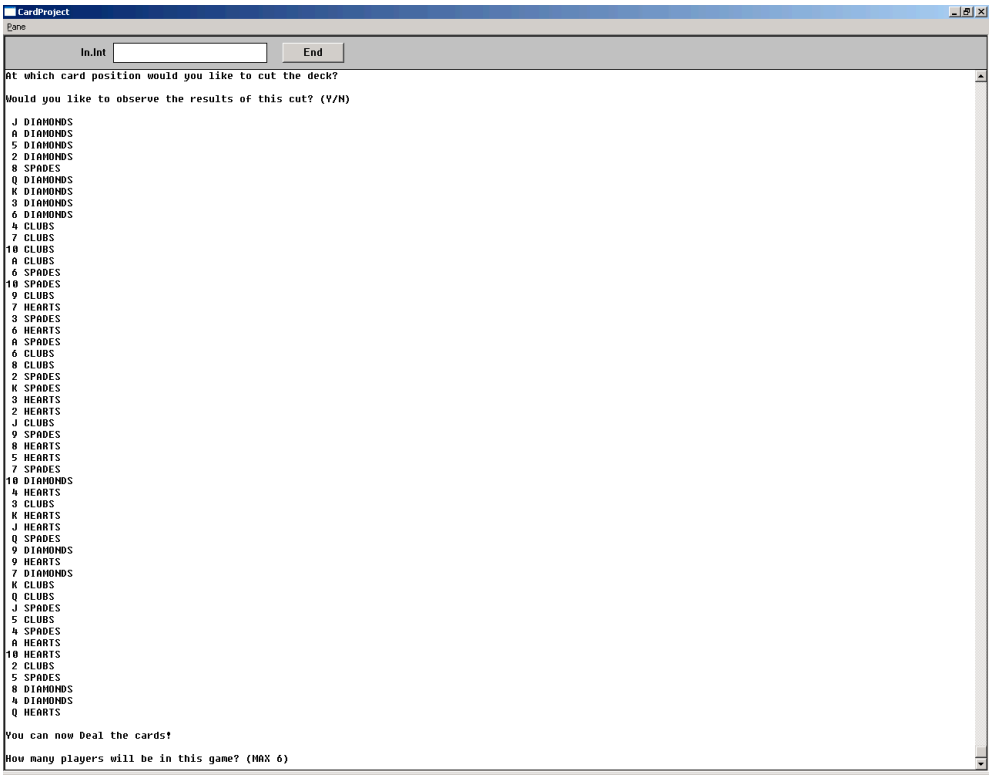

Figure B.10 – Results of a combined shuffle and cut.

Appendix C

Code Listing