Writing Card Games: An Early Excursion into Software Engineering Principles

John K. Estell
Electrical & Computer Engineering and Computer Science Department
Ohio Northern University

1. Introduction

Card game programs are both visual and event-driven; playing cards serve as a well-recognized graphical element and the play of the game progresses through the handling of discrete user-generated events. As assignments, games are often challenging to write, but provide both a definite goal to strive for and a greater sense of accomplishment as the completed program actually does something. Along with the motivational value of such assignments, the writing of games promotes strategic thinking. A programmer must consider how to properly utilize data structures to represent the elements of the game and how to establish the necessary heuristics for evaluating the status of the game.

In the past, each card game program had to be essentially written from scratch, but what really changes from implementation of one game to the next? How does the concept of a card or a deck differ? There is a great deal of functionality that stays the same, regardless of the card game being implemented. This card game assignment is used in our third introductory programming course, where after two quarters of C++ in a text-based context, students are introduced to graphical user interfaces (GUI), event handling, and code libraries (including the Collections Framework) using Java. This assignment takes an object-oriented programming approach to the problem to determine the constituent parts of a card game, from which contracts are developed that students are asked to implement. By taking this approach, the writing of the code can be compartmentalized into classes that are easy to write and can be readily reused, leaving only a small amount of code that has to be explicitly written for a particular card game application. Additionally, multiple test programs are written during the development of the contracted classes, allowing for reliability verification as well as providing first-hand experience with class reusability. With this approach, students receive an early exposure to the essence of software engineering principles: working out the specification, design, and testing of a construct of interlocking concepts involving data sets, relationships among data items, algorithms, and invocations of functions[1]. This exposure can then serve as a foundation upon which these concepts can be expanded and refined throughout the remainder of the curriculum.

2. Design by contract and the importance of documentation

One of the major concepts to be conveyed to students is that program development is usually not performed in isolation. Due to their size and complexity, modern applications require teams of developers; accordingly, classes have to be written such that they can be understood by various constituencies: those who write the class, those who use the class, those who extend the class, and those who maintain the class. Additionally, students need to be aware that, upon entry into their first job, their place will be on the lowest rung on the career ladder, and in that position they will often be asked to implement, not design, functionality for a portion of an application. It is within this context that the topics of Design by Contract and documentation are introduced.

The concept of Design by Contract (DBC) was developed by Bertrand Meyer as part of the Eiffel programming language[2]; however, the principles can be applied to other languages such as Java. The basic premise involves improving the reliability of software systems through the establishment of contracts that define the benefits and obligations between parties; specifically, between those who write classes and those who use those classes. The contract protects both parties by stating how much should be done by the contractor on behalf of the client and by stating the limited scope for which the contractor is liable. In practice, this is done through the specification of preconditions and postconditions for every method made publicly available through a class. The set of preconditions constitute the requirements placed upon the client by the class for using the method; these are usually requirements stating restrictions on the arguments being passed to the method. The class implementing the method is only held responsible for working correctly if the caller adheres to the stated preconditions. The set of postconditions constitute promises made by the class to the client; these are usually a specification of the effects that the method will have on program data or program state. The goal of DBC is to construct a specification for a class that is simple, clear, and accurate which allows a client to easily understand and utilize the features of a class without having to worry about its implementation.

Java provides a mechanism for incorporating DBC information from source code through the use of Javadoc[3], a utility program distributed with the Java SDK that builds API documentation in the form of HTML files from specially embedded comments contained within the source code of the classes that constitute the project. Javadoc has been used to generate the vast majority of API documentation; in many cases, it constitutes the only documentation for a class. As it is part of the source code, responsibility for proper documentation rests with the programmer[4]; accordingly, it is incumbent upon the instructor to discuss the need for proper documentation, as bad documentation is just as severe an affront to software engineering principles as bad code is as it inhibits, and frequently prevents, code reusability. The use of documentation needs to be more than just the bare essentials that state the purpose of a method. At a minimum, beginning programmers should specify the preconditions and postconditions for each method, and provide a description of the purpose and behavior of each class and its use of or by other classes. Ideally, this documentation exercise is performed early in the coding process so as to serve as a blueprint for code development instead of the "stating the obvious" afterthought comment commonly inserted into code at the end of the development process. The use of the Javadoc `@param` tag allows for the easy declaration of most of the preconditions, in terms of declaring the purpose of each parameter and, if necessary, any constraints upon that parameter. Correspondingly, the

Javadoc `@return` tag provides a mechanism to describe the meaning of the returned value, the potential range of values and how special circumstances that may occur are reflected in the returned value. Given that the card game assignment described here is targeted toward first-year students, attempting to have them incorporate comprehensive documentation could prove to be counterproductive; in short, it is not a fun exercise. Therefore, for the purposes of this assignment, the instructor develops stub code for the fundamental classes (`Rank`, `Suit`, `Card`, `Deck`, and `Hand`) that incorporate Javadoc-formatted comments; the Javadoc program is used to generate the contracts to be provided to the students. The documentation contained therein is kept minimal so as to provide classroom discussion fodder and for encouraging students to enhance the specifications of the method behaviors. The next four sections[5] provide an overview of the contracted classes.

3. The `Rank` and `Suit` classes

There are two properties that describe a playing card. The *suit* of a card refers to one of four possible sets of playing cards in a deck: clubs, diamonds, hearts, and spades. The *rank* of a card refers to the name of a card within a suit: ace, two, three, four, five, six, seven, eight, nine, ten, jack, queen, and king. Traditionally, the rank is used to specify the ordering of cards within a suit, *e.g.* the two comes before the three, and the jack comes before the queen. The combination of suit and rank describes each card found in a standard deck of playing cards. To express the rank and suit values, students are instructed through the contract specifications for both the `Rank` and `Suit` classes to implement the values (such as `ACE`, `FIVE`, and `JACK`) as publicly accessible symbolic constants defined through the instantiation of the appropriate private constructor. This methodology allows the necessary values to be readily available without the headache of erroneous values potentially being instantiated by the client; it also gets away from the use of explicit "magic numbers" to represent conceptual values such as a club or a jack.

As cards must be evaluated in order to determine a winning hand, and are often sorted for the purpose of displaying a hand, the `Comparable` interface is implemented for both classes. The `compareTo()` method utilizes an unmodifiable public list of values of the symbolic constants for each class, constructed in ascending order of value,. The comparison is performed by calculating the difference between the list indices of the two values being compared and returning that result. The `Rank` class presents a special case in that it is commonplace for card games to specify that either the rank of king or ace constitutes the highest rank within a suit. Accordingly, the static methods `setAceHigh()` and `setKingHigh()` in the `Rank` class allow comparisons to be performed with whatever rank is appropriate for the implemented game being set by the client as the highest ranking card. The implementation of the public list of values for each class also allows for iteration over both lists when one instantiates a set of cards to be placed into a deck.

4. The `Card` class

For uniquely describing a card, it is sufficient to use the rank and suit properties. However, in order to work in a visually-oriented environment, a third property is required: the image containing the graphical representation of the card. One of the problems with GUI implementations of card games is finding card images that are not encumbered by copyrights; fortunately, there is a set of card images available through the GNU General Public License[6]. The presentation of this resource to students allows for a discussion of both computer ethics issues and an introduction to the Free Software Foundation and the ongoing discussion between free and proprietary software.

The format of the filenames for these images is such that the process of reading in the images can be automated. All of the images are stored in individual files using filenames in the form of:

RS`.gif`

where ***R*** is a single character used to represent the rank of the card and ***S*** is a single character used to represent the suit of the card. The characters used for ***R*** are: `'a'` (ace), `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, `'9'`, `'t'` (for 10), `'j'` (jack), `'q'` (queen), and `'k'` (king). The characters used for ***S*** are: `'c'` (clubs), `'d'` (diamonds), `'h'` (hearts), and `'s'` (spades). Two additional cards are also available: `b.gif` (back of card) and `j.gif` (joker). To assist with the generation of filenames, the symbolic constant objects defined in the `Rank` and `Suit` classes store the characters associated with each rank and suit value. The static `getFilename()` method refers to these values and bases its generation of the filenames according to the rules specified above. The instantiation of a `Card` object is thereby based on not only the rank and suit values, but also with a reference to the appropriate image file.

The `Card` class implements the `Comparable` interface for the purpose of sorting a hand of cards; its `compareTo()` method, in turn, relies on the implementation of the `Comparable` interface in the `Rank` and `Suit` classes. As it is preferable in some games to sort the cards in rank-major order whereas in other games suit-major order is preferred, the `Card` class has two static methods, `setRankMajorSort()` and `setSuitMajorSort()`, that allows the client to specify the appropriate ordering method for the card game being implemented.

5. The `Hand` class

The `Hand` class represents the basic functionality of a hand of cards. Those operations that are normally conducted upon a hand, such as adding or removing cards, are supported; however, the evaluation of the cards contained in the hand is defined as an abstract method. This allows code common to the implementation of a hand in various games to be written once and reused as needed. The code specifically required for the evaluation of a hand in a particular game is developed within a class extended from `Hand` by providing a definition of the `evaluateHand()` method; this method can then be accessed either directly or via a superclass reference when comparing or evaluating hands.

6. The `Deck` class

The `Deck` class serves as a container for `Card` objects, and possesses the functionality of a typical deck of cards. When instantiated, a `Deck` object is deliberately left empty; the client is required to populate the deck with cards via iteration on the sets of rank and suit values. This is done because the nature of what constitutes a deck of cards differs between games; for example, a "standard" deck contains 52 cards, with each card present only once, whereas a pinochle deck contains 48 cards consisting of an ace, king, queen, jack, 10, and 9 in each of the four suits, with two of each card being present. The population of a `Deck` object is performed by iterating over the range of rank and suit values appropriate to the game being implemented, instantiating the card, then adding the card to the deck. For each card, the relative pathname of the image associated with those values must be generated; this is usually a combination of the name of the directory storing the card images and the filename of the specified image. Once the deck has been populated with cards, it is normally shuffled, then used to "deal" cards. It is important to emphasize to students throughout the curriculum that the performance of actions should not necessarily be implemented literally; dealing from a deck of cards provides an excellent example. To deal cards from a deck, the literal approach would remove one card from the deck, and add that card to a player's hand. However, this approach adds needless complexity to the implementation of the game, as the transition from the play of one hand to the next would require the collection of all cards from discard piles and player's hands. Instead, a instructor can show that, by treating the set of cards in the deck as immutable after initialization and allowing the ordering of these cards to be mutable through invocation of a `shuffle()` method, the drawing of cards from a deck can be easily implemented by using an integer index referring to the "top of deck" location in the list to pass back a `Card` reference for the player's hand. At the end of each hand, all "dealt" references to the cards are simply discarded, and the deck is restored through invocation of the `restoreDeck()` method, which simply resets the integer index to its initial value.

7. Testing the classes

The need to test the correctness of one's code cannot be overstressed; fortunately, this assignment provides opportunities to systematically check the implementation of the various classes. Before the actual card game is written, students are instructed to write three programs that test various features of the contracted classes. Students are provided with written specifications for each of these programs, and detailed images of the user interface are included with the program specifications to provide the appropriate look and feel of each test application. In addition, the web site for the assignment contains fully-implemented executables of all three test programs for the students to play with; while this is not a normal situation that one encounters in the development of software engineering products, the ability to interact does provide a bridge of understanding for the first-year student in need of reassurance. The design of the test programs is such that students are exposed to an iterative development process, where one builds upon the successes of the previous test programs, all of which serves as a stable foundation for the development of the actual card game program. This methodology acts as a quality assurance program that focuses on discovering implementation bugs early on in the process through the use of test programs employing simple constructs that permits students to focus on the contracted classes as the source of any observed defects. The result of this approach allows students to have confidence in the performance of the contracted classes when writing the actual card game, so when an error occurs, the focus of the debugging efforts can be localized to just the GUI, event handlers, and hand evaluation algorithms developed for the game. The concept of finding defective code early should be mentioned to the students as part of the overall discussion for the assignment, ideally in such a way so as to contrast the relative levels of difficulty between finding an error where the scope is limited to a small amount of untested code and finding the same error where the scope encompasses the entire program.

The first test program, `CardDeckDemo`, is used to exercise the implementation of the `Card` and `Deck` classes. The program uses a simple interface, shown in Figure 1. A `JLabel` displays both the image and the name of the card just drawn from a deck; by displaying the rank and suit information of the card, it is easy to verify that the correct image has been associated with the card object. Two `JButton`s are used to handle user interactions: one for drawing a card from the deck, and the other to both restore and shuffle the deck. A second `JLabel` is used to indicate the number of cards remaining in the deck; when the deck is empty, the "Draw a card" button is disabled until the "New & shuffled deck" button is pressed. This program allows the student to verify that all of the cards were instantiated correctly and that the basic deck operations of dealing, restoring, and shuffling a deck, along with the detection of an empty deck, are properly implemented.

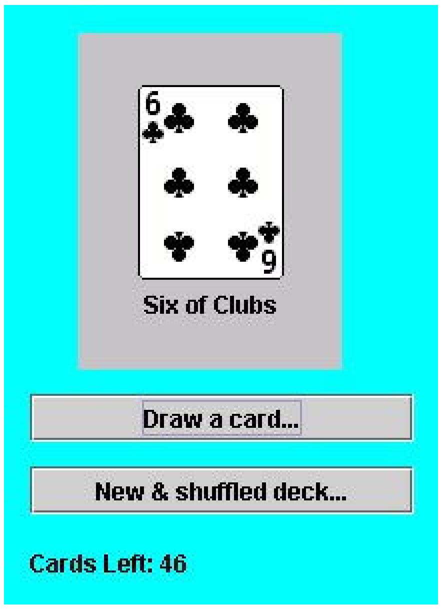

Figure 1. Snapshot from `CardDeckDemo` program.

The second test program is the "Dumb Game" card game, which is exactly that, at least when viewed as a game. However, its real purpose is to allow the student to write a simple extension of the `Hand` class to implement the evaluation algorithm and to exercise some additional methods in the other contracted classes. The implementation of this program is relatively simple, as most of the code has already been written. Only two additional source code files need to be developed: `DumbGameHand`, which is a subclass of `Hand` where one provides the definition for the `evaluateHand()` method, and `DumbGame`, which contains the definitions of the GUI and the event handlers for this card game.

The operation of the game is straightforward. Eight cards are drawn from a full deck of cards and placed into the player's hand; the cards are displayed on separate `JLabel`s with both their graphical image and their name. Each card is worth its displayed pip value (ace = 1, two = 2, *etc.*) in points, with face cards worth ten points. The value of a hand is equal to the summation of the points of all the cards held in the hand; this value is displayed to the user. Figure 2 presents the initial display of the program's user interface; note that in this instance the hand is deliberately dealt from an unsorted deck to further verify the proper instantiation of the cards and construction of the deck.

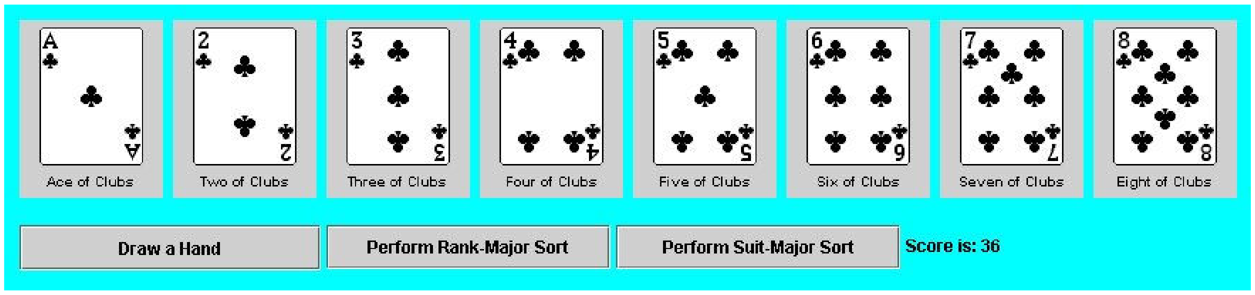

Figure 2. Initial display for `DumbGame` program.

When the "Draw a Hand" button is pressed, the generated event is used to restore the deck to its original number of cards, shuffle the cards, draw a new hand, and display the unsorted result to the user, as shown in Figure 3. Upon the drawing of a hand, the `evaluateHand()` method is invoked, where an integer equal to the total number of pips is returned and displayed to the user. The operation of this method can be verified by manually counting the pips shown on the displayed card images.

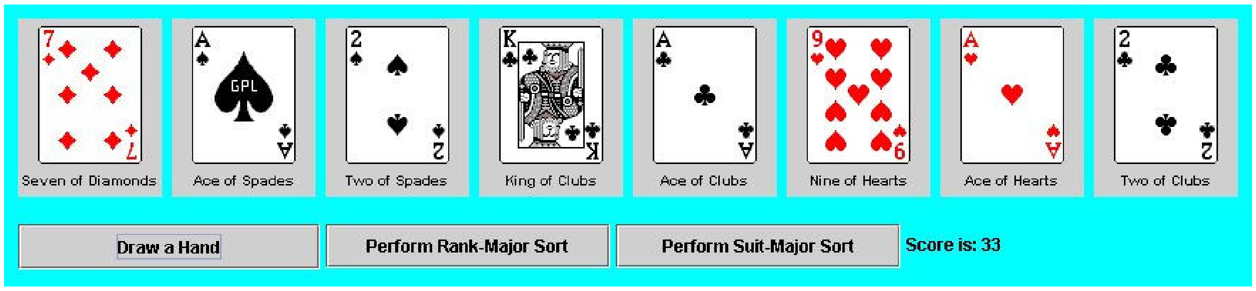

Figure 3. A new hand is drawn.

The player now has the exciting options of sorting the drawn cards in either rank-major or suit-major order through use of the "Perform Rank-Major Sort" and "Perform Suit-Major Sort" `JButtons`. This allows for the verification of the `Comparable` interfaces for the `Card`, `Rank`, and `Suit` classes. The results of pressing these two buttons are shown in Figures 4 and 5.

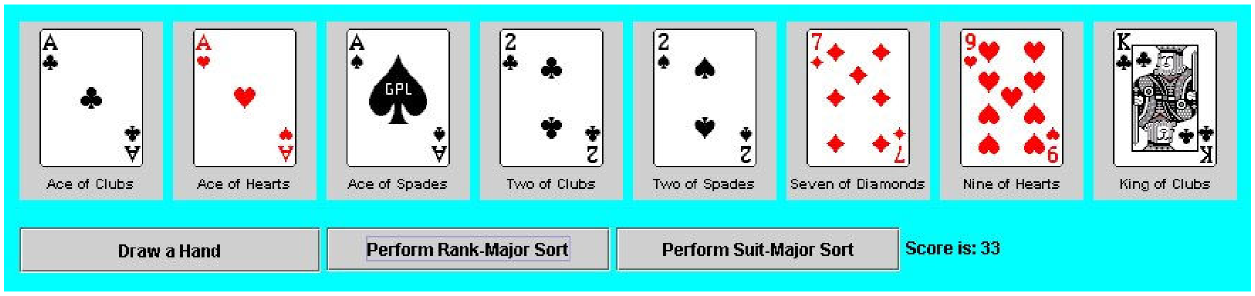

Figure 4. The hand is displayed in rank-major order.

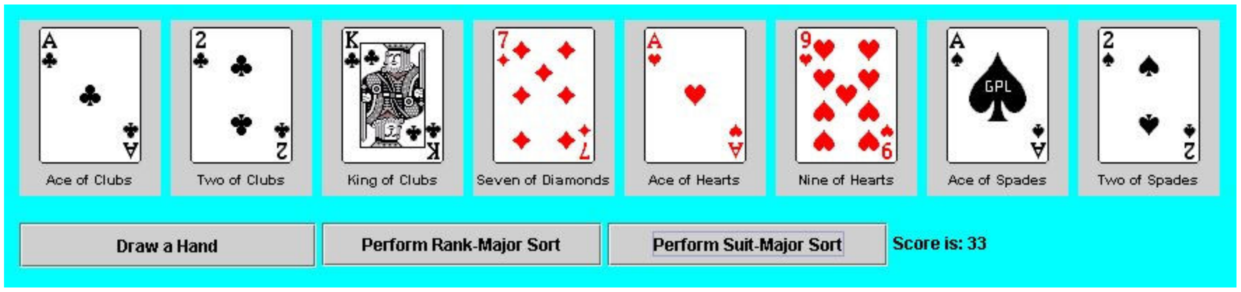

Figure 5. The hand is displayed in suit-major order.

The third test program demonstrates the power of the object-oriented programming paradigm by having the students write Beginner's Blackjack, which is a simplified version of the classic card game. In this version of Blackjack, only two cards are dealt. The scoring of the hand is according to the rules of Blackjack: the face cards are each worth ten points, and the other cards are worth the number of pips displayed, save for the ace. In regular Blackjack, the ace can be worth either one or eleven points; however, in this version, the ace is always worth eleven points as it is impossible to go over 21 with this simplified format. The two cards drawn from the deck are displayed on `JLabels` and the resulting score is also shown on a `JLabel`. Each time the "draw a hand" event occurs, the deck is restored and shuffled, after which two cards are drawn from the deck and placed into the player's hand. Figure 6 illustrates the look of a typical Beginner's Blackjack program:

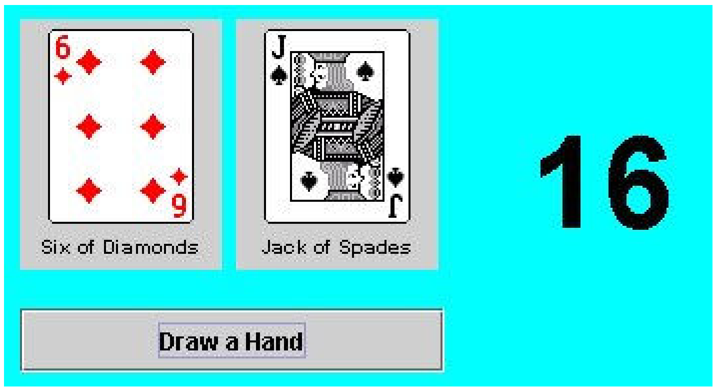

Figure 6. Snapshot of Beginner's Blackjack program.

The object of Beginner's Blackjack is to score a blackjack; when this happens, the display is modified such that the score has an exclamation point appended to it and is displayed in a different color. Figure 7 shows how the program displays a blackjack to the user:

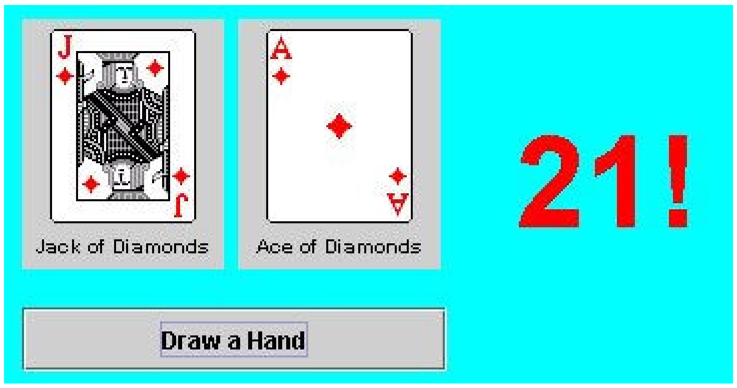

Figure 7. Scoring a blackjack.

To implement this game, all that is needed is to write the `evaluateHand()` method for a subclass of `Hand`, and the source code that implements both the graphical user interface and the "draw a hand" event. The simplicity of this implementation drives home, through a hands-on example, the concept of reusable code in a way that any student can relate to.

Once students demonstrate that their three test programs that exercise the methods contained in the contracted classes have been successfully implemented, they are allowed to begin work on the actual card game. From the test programs they have learned what code can be reused and what has to be written specifically for the new game, and that they can be confident in the correctness of the methods in the contracted classes that are about to be reused.

8. Implementing the actual card game

At this point students can implement an actual (*i.e.* authentic) card game. There is great latitude on the part of the instructor as to how this portion of the assignment is conducted in that students can be instructed to implement a specific game, select from a set of "approved" games, or select any game, subject to approval. The United States Playing Card Company maintains a card game rules archive[7] to which the students can be referred in order to minimize the amount of instructor effort required to convey the rules of a particular game to students. It is not a good idea to give carte blanche to the students with the selection process, as there will be some students who will take advantage of the opportunity to implement games such as "War," which in some ways is a more simplistic game than the test programs. A reasonable heuristic to use when determining acceptable card games for this assignment is that the development of the game should result in the implementation of an "interesting" hand evaluation algorithm and "meaningful" interaction with the GUI. One approach used by the author is to base the possible number of points available on the amount of functionality, in terms of user interaction, present in the game, and to ban outright games with overly simplistic evaluation algorithms. Programs that implement card games where there is no interaction with the cards whatsoever are discouraged by weighting them at the low end of the scale. Games that involve interaction at the level of indicating whether to draw another card, such as Blackjack, are rated at the middle of the scale. Games that involve actual selection of cards (*e.g.* Poker, Euchre, Hearts) in order to play the game are weighted at the high end of the scale. Extra credit is made available for programs going beyond "traditional" implementations. The card game rules archive contains description of variations to traditional games that, in order to implement, result in a greater amount of complexity. A currently popular example is Texas Hold'em Poker, where such elements as wagering and artificial intelligence agents representing other players at the table are required. In all cases, students are required to submit a design document prior to the development of the actual card game that states what game is to be implemented and documents any proposed additions to the basic functionality of the game.

9. Results and conclusions

For those readers who are interested, resources for this assignment, including source and stub code, documentation for the classes, card images, and demonstration applets for the three test programs are available at the Nifty Assignments web site[8]. Please note that this is not a typical assignment where one writes essentially disposable code for the instructor. Students participate in an iterative process where a realistic schedule is imposed for the delivery of test programs designed to verify the correctness of contracted classes, upon which the specification for an actual card game can be implemented based upon reusable and reliable code. Card games provide interesting programming challenges that can be solved through the application of software engineering principles; accordingly, this assignment captures the essence of the software engineering process as applied to a legitimate programming problem, but is tempered so as not to overwhelm. Through this assignment, students receive early exposure to many software engineering principles, which is important to instructors. However, as reflected in their student evaluation comments, one of the important factors of this assignment to students is that the final program actually does something interesting once it is completed. While students will always complain to some degree about the workload required for implementing any program, they want to have meaningful assignments, not busywork, handed out to them – and if the resultant deliverable is fun to use, so much the better. Providing positive modes of motivation is never easy for an instructor, so wrapping up various learning outcomes into an attractive package for the students to open is of benefit to everyone. Variations on the card game assignment have by used by the author in his first-year programming classes for the past several years, and it remains one of the favorite and more memorable assignments conducted by his students.

Bibliography

1. F. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer, vol. 20, no. 4 (April 1987), pp. 10-19.
2. B. Meyer, "Applying 'Design by Contract'," IEEE Computer, vol. 25, no. 10 (October 1992), pp. 40-51.
3. Javadoc Tool web site. Available online: http://java.sun.com/j2se/javadoc/
4. B. Goetz, "Java theory and practice: I have to document THAT?", August 2002. Available online: http://www-106.ibm.com/developerworks/java/j-jtp0821.html
5. J. Estell, "Teaching Graphical User Interfaces and Event Handling through Games," Proceedings of the 2004 American Society for Engineering Education Annual Conference. Available online: http://asee.org/acPapers/2004-862_Final.pdf
6. "About the Cards" web page. Available online: http://www.waste.org/~oxymoron/cards/
7. United States Playing Card Company Card Game Rules Archive. Available online: http://www.usplayingcard.com
8. J. Estell, "The Card Game Assignment," Nifty Assignments web site, March 2004. Available online: http://nifty.stanford.edu/2004/EstellCardGame/index.html

Biographical Information

JOHN K. ESTELL became Chair of the Electrical & Computer Engineering and Computer Science Department at Ohio Northern University in 2001. He received his BS (1984) degree in computer science and engineering from The University of Toledo and received both his MS (1987) and PhD (1991) degrees in computer science from the University of Illinois at Urbana-Champaign. His areas of interest include simplifying the program outcomes assessment process, interface design and embedded applications. Dr. Estell is a Senior Member of IEEE, and a member of ACM, ASEE, Tau Beta Pi, Eta Kappa Nu, and Upsilon Pi Epsilon.