TEACHING GRAPHICAL USER INTERFACES AND EVENT HANDLING THROUGH GAMES

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

Introduction

The introductory programming sequence for both computer engineering and computer science majors at Ohio Northern University can be summarized as follows. The first course covers the concepts of sequence, iteration, and selection. The second course explores the object-oriented programming paradigm. Finally, the third course reinforces the object-oriented programming paradigm and introduces the graphical user interface (GUI) and the related concept of event handling. After two courses with a text-based focus, the introduction of visual components in the third course provides an opportunity to excite students about programming. However, in order to reach today's students, one must understand that their perception of computers is different than that once held by today's faculty when they were entering the profession. Many of our students have their conceptual images of computers formed primarily through their interactions with video games and GUI-based applications. Given this context, the use of games is an effective motivational tool as students now have the opportunity to study that which they easily relate to. Most games are both visual and event-driven; usually there is a graphical element such as playing cards or a game board, 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. Furthermore, by providing extra credit opportunities for the implementation of additional game features, students become very involved in their programming, helping them to learn the concepts taught in the course – and often to learn advanced concepts on their own. Along with the motivational value of such assignments, the writing of games promotes strategic thinking [1]. Students must consider how to properly utilize data structures to represent the physical elements of the game and how to establish the necessary heuristics for evaluating the status of the game. As getting the logic right is required for the game to play properly, and as having a correctly working game provides the student with a significant amount of positive feedback, the strategic aspect of writing game programs forces students to pay greater attention to the construction of their code than they would otherwise be experiencing. There is another valid reason for the early introduction of GUIs and event handling. Frankly, while still useful, the teletype approach to programming cannot fully prepare students with the skills required for 21st century software development [2]. Modern platforms are increasingly graphically oriented and event driven, and so software development in this area needs to be addressed. Furthermore, the tools for such development have grown to the point that much of the "gruntwork" is handled either internally by the system or automatically generated by the IDE used to write the program. In short, the complexity of such a task has diminished to a point where a first-year student can successfully accomplish it.

Platform

The Java programming language was adopted for this course as it lends itself well to the subject. Java is an object-oriented language featuring a rich built-in library of routines, including component libraries for the development of graphical user interfaces. The implementation of any sufficiently interesting GUI tends to result in pages worth of tedious code. However, most Java development systems (such as JBuilder and NetBeans) allow for automatic GUI code generation through visual drag-and-drop mechanisms. The event model is relatively simple, with some development systems supplying interaction wizards to assist with code generation. Swing, which is the Java Foundation Classes' GUI component library, is robust yet well designed and allows for the easy inclusion of graphical images contained in GIF or JPEG format. The built-in pixel-oriented graphics context allows for easy rendering, and provides an option for the incorporation of introductory computer graphics material into the course curriculum. Finally, assignments can be developed either as applets orexecutable JAR file applications, allowing the program to be easily assessable to the instructor.

Initial Concepts: Labels, Buttons, and ActionEvents

Traditionally, one introduces students to the design of graphical user interfaces using components that are simple to work with. The simplest component to work with is the label, which is normally used for static output displays but can also be used for dynamic displays via event-driven constructs. The use of Swing's JLabel provides additional design flexibility through its ability to display images in addition to text. All that is needed is either a GIF or a JPG file to be loaded into memory as an instance of an ImageIcon class, followed by a call to the setIcon() method of the appropriate JLabel:

```
ImageIcon d2 = new ImageIcon( getImage(
   getCodeBase(), "die2.gif" ) );
dice1Label.setIcon( d2 );
```

For input, the simplest approach is to use a button, as user input is thereby limited to just clicking the mouse while the cursor is over the component. As with the JLabel, the JButton allows for the display of both text and images, again through use of the setIcon() method. However, in order for a button to be properly used, the concept of event handling as implemented in Java must be covered. Essentially, whenever the user presses a key, moves the cursor, or clicks a mouse button, an event occurs. In order for a component to be associated with an event, the appropriate interface needs to be implemented and the specific event must be registered such that the component becomes an event listener. Buttons use action events; in order for user input to be recognized, the ActionListener interface must be implemented by the program and the addActionListener() method must be applied to the button object. When a registered event is received, program control is transferred to the appropriate actionPerformed() method, which contains the code that handles the actual event.

While the above is sufficient for an introduction to GUI programming, two other items must be introduced at this time in order for a student to effectively implement a game program that is worth playing. Random numbers are a vital part of game programming, as it is at the heart of making many games appear realistic, such as those involving dice or cards. Random numbers are easily generated through appropriate method calls upon an instantiation of a Random object. As an example, if one were to implement a dice game, each roll of a die would be simulated through the invocation of the nextInt() method on the object, with an integer parameter passed to specify an exclusive upper bound and an additive translation employed to finalize the result to the familiar cubic die values of one through six:

```
Random generator = new Random();
int dieValue = generator.nextInt( 6 ) + 1;
```

The other item that is useful to introduce here is the use of timers. The Timer class, which is part of the Swing hierarchy, allows for the implementation of program-controlled GUI interaction, which can be used to implement such things as animation or constrained interactions (e.g. providing the user with a limited amount of time to provide a response). It is recommended that Swing-based timers be used for GUI-related tasks as Swing timers all share the same pre-existing timer thread and the

GUI-related task is automatically executed on the event-dispatching thread. [3] Timers generate action events just like buttons; accordingly, it is easy to introduce the material early on, once the concept of event handling has been covered. A timer is operated by first instantiating an object featuring the specification of a delay in milliseconds between events and the location of the action event handler code. The start() method is invoked upon the timer object, upon which action events are fired at regular intervals. If necessary, Timer objects can be set up in a "one-shot" configuration, turned on or off as necessary, or even have the period between events varied as needed.

An example program suitable for an introductory game assignment is "Shut the Box!" In this game, the player has nine tiles, enumerated 1 through 9, which are initially face up. Upon rolling the dice, the player lays down any combination of tiles to match the total value shown on the dice. The player continues to roll until no more tiles can be laid down. The player is said to have "shut the box" if all of the tiles are turned down. The total of the remaining tiles is the final score, with the player having the lowest total being the winner. The implementation of this game, a screenshot of which is shown in Figure 1, is straightforward.

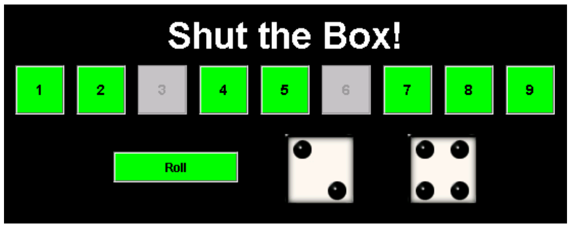


Figure 1. Screenshot of "Shut the Box!" applet.

The dice are implemented using integer variables; to roll the dice, the user clicks the "Roll" JButton. The event causes the values of the die to be set via calls to the nextInt() method of a Random object; these values are then used to access dice images stored in an ImageIcon array such that the value of the array index corresponds to the number of dots displayed on the die. Finally, the selected images are used to display the rolled dice on two JLabels. The tiles are implemented also using buttons. When a tile is selected, the background color is changed to show that it has been "laid down." When the sum of the tiles laid down for the current turn equal the total value of the dice, those tiles are then disabled and the next turn commences. Options that can be included are a "new game" button and a label to show the low score.

Incorporating Text

There are many games that require text-based input and/or output; it is therefore appropriate to introduce such GUI widgets as text fields, text areas, and choice buttons in a game-playing context. The text field is often portrayed as useful for both input and output; however, labels are more appropriate for text output as they are rendered without the unsightliness of the surrounding box that visually characterizes a text field. When used for input, an action event has to be registered for the text field. Swing provides the JTextField component, which allows for any string input. If the input needs to be restricted, then testing must be performed as part of the action event handling routine. An alternative approach, available as of Java Release 1.4, is to use a subclass of JTextField called JFormattedTextField, where the programmer can specify the legal set of characters that can be entered into a text field.

An example program using text fields for input and labels for output is the "Hi Lo Game," a screenshot of which is provided in Figure 2. The purpose of the game is to guess a randomly generated integer that ranges between 0 and 99, inclusive, in six or fewer tries. If the answer is correctly guessed, a label is used to indicate a successful outcome to the player. After each incorrect guess, the player is informed as to whether the guess was too high or too low relative to the correct answer, and (unless it was the final guess) the text field for the next guess and its corresponding prompting label are made visible to the player. If the answer is not correctly guessed within the allowed number of guesses, then a label is used to inform the player that he/she lost and also to show what the correct answer was. At the end of the game, a button is displayed to offer the player the opportunity to start a new game; this button is invisible the rest of the time.

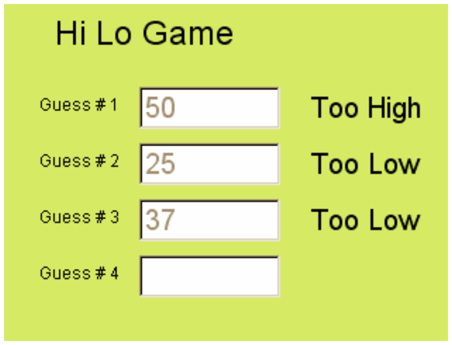


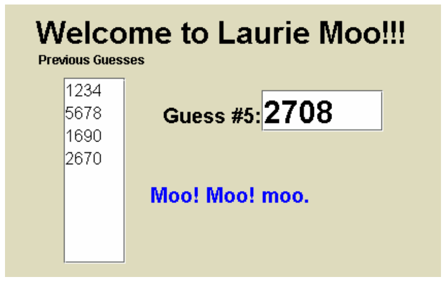
Figure 2. Screenshot of "Hi Lo Game" applet.

For this game, students must take the input as a string from the text field and convert it into an integer through use of the parseInt() static method of the Integer class. This method will throw a NumberFormatException if the string does not contain a parsable integer; this allows students the opportunity to incorporate a try-catch block into their code to idiot-proof the input and respond accordingly via the information label to the right of the text field. To promote the concept of classes, students are required to write the class RandomInt, to be stored in the file RandomInt.java, for dealing with the random integers to be generated and processed for this assignment. Methods to be supported are:

- RandomInt(int maxIntValue) – a constructor that will instantiate a RandomInt object with a "correct answer" value in the range [0, maxIntValue] having been generated.
- int getCorrectAnswer() – an inspector that returns the "correct answer" integer.
- boolean isCorrectGuess(int guess) – returns true is the passed guessed integer is the "correct answer" integer.
- boolean guessIsTooHigh(int guess) – returns true if the guessed integer is greater than the correct answer.
- boolean guessIsTooLow(int guess) – returns true if the guessed integer is less than the correct answer.

Another game that utilizes random numbers and text fields, and includes the use of text areas, is the "Laurie Moo" program, which is a variant of the classic Mastermind game (this version features a stuffed toy cow named Laurie Sue). The game, a screenshot of which is presented in Figure 3, is similar to the Hi Lo Game: you have ten attempts to guess a four-digit number randomly generated by the game. However, the feedback from this program is different. For each digit correctly specified (both value and position), you get a "Moo!" displayed by the program. For each digit specified that is not in the correct position but is in the solution, you get a "moo." displayed by the program. If none of the digits match in any way, then the message "all you hear are cowbells" is displayed. If the number is correctly guessed then, in addition to all of the "Moo!" strings being printed out, you also get a "LaurieMOO!!!" displayed as a "reward." If the user fails to guess the number after ten attempts, the message "Boo hoo -- no LaurieMOO" is displayed. Adventurous students in search of extra credit can enhance the Laurie Moo experience through the addition of sound effects via the Java Sound API. The text area in this program is used to display the previously unsuccessful inputs. Under Swing, the JTextArea is normally embedded within a JScrollPane container so that, if necessary, the contents of the text area can range outside of the provided viewing area space allotted to it on the applet yet still be accessible through use of scroll bars.

The choice button is a practical way of providing a set of input options to a player without having to dedicate a large amount of real estate on the GUI. Accessed through the JComboBox class in Swing, the choice button provides a list of items from which the user can make a selection; furthermore, this list can be dynamically updated during the execution of the



Welcome to Laurie Moo!!!

Previous Guesses

1234
5678
1690
2670

Guess #5: 2708

Moo! Moo! moo.

Figure 3. Screenshot of "Laurie Moo" applet.

program. An example program that utilizes the choice button is the "Dice Game" applet; the screenshots from two implementations of this program are presented in Figure 4. The dice game is very similar to Yahtzee, but with the student implementing sufficient changes such that there would be no trademark infringement (such as not using the word Yahtzee and altering the scoring algorithm). For this assignment, students were asked to implement an applet such that its footprint would not exceed 100 by 150 pixels, with the provision that it was permissible for the choice button to exceed the prescribed boundaries when activated. For both applets shown below, dice images are placed onto buttons; clicking on the buttons allows one to either hold or roll selected dice on the next roll. As the JButton allows for both text and images to be displayed, it is simple to include the "HOLD" text indicating an individual die's status, and there are methods that allow for the positioning of the text both relative to the image and relative to the component. For both implementations, scoring occurs by the selection of the appropriate category contained in the choice button; the text for that item is then updated so that its use can be shown on future turns.

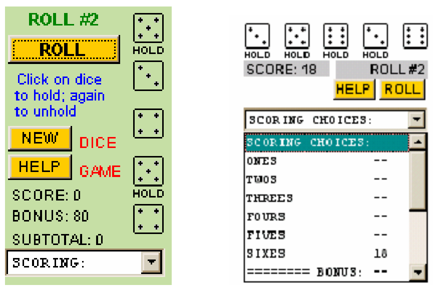


Figure 4. Screenshots from two dice game implementations

The Cursor and MouseEvents

There are times where, in order to promote a certain look and feel for a program, a programmer either does not want to or cannot utilize the "canned" input method of clicking upon a button component. In these situations, mouse events are used. The application of mouse events can provide a great deal of flexibility in determining the overall operation of a program, as essentially anything a user can do with the mouse can be trapped, just as long as the widget listening for a mouse event is derived from the Component class. Because of the nature of mouse interactions, there are two different interfaces available. The MouseListener interface is used to deal with the pressing (mousePressed), clicking (mouseClicked), and releasing (mouseReleased) of a mouse button; it is also used to detect when the cursor has been moved onto (mouseEntered), or off from (mouseExited), a registered component. It should be noted that, when implementing this interface, all five methods must be defined in the program, even if only one method is actually needed. This is handled either with an empty method body for each unneeded method or by an extension of the MouseAdapter class, where one inherits all empty methods and writes overriding methods for just those whose functionality is required. The MouseMotionListener interface is used to deal with both the regular movement (mouseMoved) of the cursor and the dragging (mouseDragged) of the cursor by moving it while a mouse button is pressed.

Whenever a registered activity with a mouse is handled, a MouseEvent object is provided to the method. This object provides valuable information regarding where the cursor was located when the event occurred. For some interactions, such as the ability to select from a set of images being displayed on individual labels, using the getSource() method will provided sufficient information as to which component was the target of the interaction. If a specific location is needed, the X and Y coordinates of the cursor location relative to the registered widget can be obtained through application of the getX() and getY() methods upon the passed MouseEvent object. This approach is used to implement the Magnetic Poetry applet shown in Figure 5.

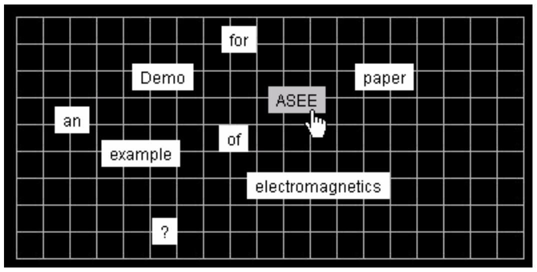


Figure 5. Screenshot of Magnetic Poetry applet

Each word is created through the instantiation of a label, which is then placed onto the applet. The label to be moved is selected by clicking on it; the cursor changes to a hand and the background color is set to gray to indicate its selection. At this time, the initial location of the cursor relative to the label is stored. Upon the dragging of the cursor, the new location of the cursor relative to the label is obtained, from which the change in the x and y values of the cursor location is calculated. The x and y coordinates for the location of the label relative to the applet are obtained, and the Δx and Δy values from the previous calculation are then added to these values to form the new location for the label, which is then moved through the invocation of the setLocation() method. In this way, the cursor appears at the same relative location on the label during the dragging process. When the mouse is released, the cursor and label background color revert to their original status. An advanced implementation can be performed through using a layered pane to contain the magnets. As the layered pane constitutes a container that can be placed onto the applet as a component, it can be sized such that it does not take up all of the applet's real estate. This allows other components to be placed onto the applet, such as a text field that allows the user to enter specific words; when an action event for this text field is received, the applet generates a new word tile through the instantiation of a label object. However, in this implementation the label is not registered to handle mouse events; instead, the layered pane is registered. By using the X and Y coordinates from the passed MouseEvent object, the getComponentAt() method can be invoked on the layered pane object to determine what component, if any, lies underneath the cursor. Once selected, the label is moved to the front of the layered pane, which allows it to be placed, if desired, on top of or overlapping another label.

Organization through Panels, Layouts, and the Collections Framework

To create more sophisticated GUIs, additional tools and techniques are needed. First, all containers have an associated layout manager, which are provided to arrange GUI components in a particular format. Several layout managers are provided, such as FlowLayout, BorderLayout, and GridLayout; a container's layout is defined using the setLayout() method. Typically, when designing a GUI, the programmer wants the various components to be placed at a specific location. In order to accomplish this, the layout manager for the program is disabled by passing a null reference to setLayout(). For most IDEs featuring a graphical GUI editor, this code is automatically generated for the programmer's benefit. However, there are instances where the use of a layout manager can be an appropriate design decision and a considerable time saver.

A panel constitutes the simplest container class in Java. Panels can be effectively used to compartmentalize one portion of a GUI display so that the components placed within the panel are organized according to the specified layout manager for that panel without affecting the layout of the other components of the GUI. A typical example where a panel is useful involves the use of game boards, where one is often faced with placing identically shaped labels into a two dimensional array. Under the Swing hierarchy, one usually creates an extension of the JPanel class; the constructor method for this class will set the layout of the panel through use of a GridLayout object, where the number of rows and columns for the game board are specified.

Labels are then instantiated (as part of an array of JLabel objects) and added to the panel in row-major order, starting in the upper left corner of the panel. Each label is registered through invocation of the addMouseListener() method so that, when a mouse interaction occurs within the panel, the appropriate label can be selected.

Finally, a collections framework is often used in conjunction with games in order to facilitate the manipulation of data using conceptual abstractions. A collections framework presents a unified architecture for working with collections of data, providing both hierarchical reusable data structures to represent collections and polymorphic algorithms for performing useful computations. Given that this course is taken prior to a formal data structures or algorithms course, expecting students to write their own sorting, searching, and shuffling routines is both asking much of the student and detracts from the implementation of the game program under consideration. By introducing a collections framework at this stage, students can utilize various routines from an abstract conceptual perspective without needing to deal with the actual implementation details. Under the Java Collections Framework, a Collection represents a group of objects, and serves as the root of the collection hierarchy. Within this hierarchy are a variety of elements such as lists, sets, and maps. From a gaming perspective the List interface, which provides an ordered collection, is the most important derivative of the Collection interface. Lists allow for positional access, searches, list iteration, and the specification of range views that allow for a subset of a list to be processed.

As an example, let us create a test program that uses a standard deck of 52 playing cards as a precursor to implementing an actual card game. 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. Additionally, implementing a card game provides a valuable opportunity to focus on the concepts of object-oriented programming, particularly in the areas of object development and code reuse. In the past, each card game program had to be essentially written from scratch; but what really changes from the implementation of one card 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 written. The complete card game assignment described here is available at the Nifty Assignments web site [4]; for this portion of the assignment, an object-oriented programming approach is used to determine the constituent parts of a card game. In particular, the card is viewed as an object and, by examining both the properties of a card and how cards are used, classes are designed accordingly. By taking this approach, the source 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 written for a particular application.

When describing a playing card, one commonly refers to two properties of a 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 uniquely describe a card found in a standard deck of playing cards. To express the rank and suit values, Rank and Suit classes are used, with constants such as JACK or SPADES being automatically constructed as privately instantiated static objects. This methodology allows the necessary values to be readily available without the headache of erroneous values potentially being instantiated by the client.

In order to work in a visually-oriented environment, a third property is required: the image, or graphical representation, of the card. One of the problems with GUI implementations of card games is finding images of cards that are not encumbered by copyrights; fortunately, there is a set of card images available through the GNU General Public License [5]. The format of the filenames for these images is such that the process of reading in the images can be automated. All of the standard card images are stored in individual files using filenames of the form:

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 other cards are also available: b.gif (back of card) and j.gif (joker). To assist with the generation of filenames, the constants defined in the Rank and Suit classes store the characters associated with each rank and suit value. The static getFilename() method in the Card class refers to these values and bases its generation of the filenames according to the rules specified above.

Information detailing the properties of the classes to be implemented can be provided either through Javadoc, which consist of HTML pages generated from documentation comments embedded with Java source code, or through Unified Modeling Language (UML) diagrams. An example UML description of the Card class is presented in Figure 6.

Card	Class name
cardImage rank suit	Instance variables of the class

static getFilename()
getCardImage()
getRank()
getSuit()
toString()
compareTo()
	Methods of the class

Figure 6. UML description of the Card class

The Deck class serves as a container for Card objects, and possesses the functionality of a typical deck of cards. A deck is implemented through use of an ArrayList from the Collections Framework. When instantiated, decks are empty; they need to be populated with cards via iteration on the sets of rank and suit values. For each card, the relative pathname of the image associated with the current rank and suit value combination 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. To create the cards and establish a deck, the following code fragment would be implemented in the init() method of an applet:

```
cardDeck = new Deck();
Iterator suitIterator =Suit.VALUES.iterator();
while ( suitIterator.hasNext() ) {
  Suit suit = (Suit) suitIterator.next();
  Iterator rankIterator = Rank.VALUES.iterator();
  while ( rankIterator.hasNext() ) {
    Rank rank = (Rank) rankIterator.next();
    String imageFile = directory +Card.getFilename   (
                    suit, rank );
    ImageIcon cardImage = new ImageIcon(
                    getImage( getCodeBase(),
              imageFile )    );
                Card card = new Card( suit, rank, cardImage
           );
                cardDeck.addCard( card );
              }
            }
```

Once the deck has been populated with cards, it is normally shuffled, then used to "deal" cards. When dealing cards, it is best not to implement the activity literally; i.e., to remove cards from the deck, as this often add needless complexity to the implementation of the game, such as how to deal with the transition from one hand to the next. Instead, the set of cards in the deck is treated as immutable after initialization, although the listing of these cards is mutated through invocation of the deck's shuffle() method, which is implemented as a wrapper function that calls upon the similarly-named method in the Collections class:

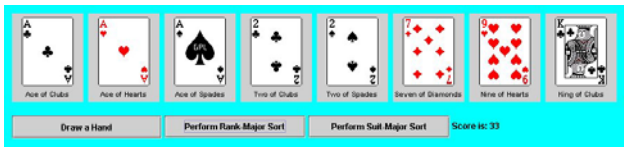
```
public void shuffle() {
   Collections.shuffle( deck );
}
```

The drawing of a card is performed by using an index variable to refer to the "top of deck" location in the list. The deck is restored to being a full deck through the invocation of the restoreDeck() method, which resets the index to the beginning of the list.

The Hand class represents the basic functionality of a hand of cards, and is implemented using an ArrayList. 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 directly or via a superclass reference when comparing or evaluating hands. The following code snippet shows how both the cards in a hand and the value of the hand are displayed to the user:

```
    myHand.sort();
    for ( int i = 0; i < SIZE_OF_HAND; i++ ) {  //
  display the hand
       Card c = myHand.getCard( i );
       handLbl[i].setIcon( c.getCardImage() );
       handLbl[i].setText( c.toString() );
    }
    scoreLbl.setText( "Score is: " +
  myHand.evaluateHand() );
```

The result of this approach can be seen in the screenshot, shown in Figure 7, of a test program applet used to verify the correctness of a student's implementation of the Card, Deck, Hand, Rank, and Suit classes.



```
Ace of Clubs   Ace of Hearts   Ace of Spades   Two of Clubs   Two of Spades   Seven of Diamonds   Nine of Hearts   King of Clubs
```

```
Score is: 33
```

Figure 7. Screenshot of card game demonstration applet

With this applet, one displays both the image and the name of the card on a label. Clicking on the "Draw a Hand" button will shuffle the deck and deal out a new hand, after which the display is updated. By clicking on one of the sort buttons, the cards in the hand are sorted by passing the hand object as the argument to the Collections.sort() static method in the order established by the compareTo() method implemented by the programmer in the Card class.

From here, students can go ahead with the implementation of an actual card game. One can either provide specifications for a particular game, the rules for which can be obtained online [4,6], or allow students to implement whatever card game they are interested in. Card games that have been implemented over the years in this course include variations on poker, blackjack, rummy, hearts, crazy eights, and Klondike solitaire. An example screenshot from one student's implementation of Klondike is presented in Figure 8.

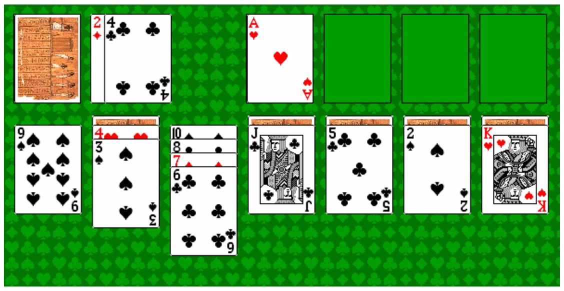


Figure 8. Screenshot of Klondike applet

Another example, which utilizes panels, layout managers, timers, and the Java Collections Framework, is the game "Concentration," based upon the long-running television game show of the 1960s and 1970s. Two contestants face a game board divided into 30 game panels (not to be confused with the panel container) consisting of 6 rows and 5 columns. Squares are enumerated starting with the number 1 game panel at the upper left, continuing sequentially across the board then down, until arriving at the number 30 game panel at the bottom right. Each game panel hides either a prize or an action card, with each card appearing twice on the board. Players alternate selecting two of the game panels by their numbers, which are then revealed. If the two prizes do not match, then the prizes are shown to the players for a limited amount of time, after which the game panels are turned back to their original (i.e. number bearing) position. If a prize is matched, the player adds the prize to his or her collection. If an action card (either "TAKE 1 GIFT" or "FORFEIT 1 GIFT") is matched, then that action must be performed. Should a player uncover a "WILD" card, the player automatically matches whatever is picked along with it. Matching both WILD cards wins the contestant a car, regardless of the outcome of the game, plus the player is allowed to select two more game panels for prizes. The car cannot be forfeited or taken. The matching prizes or actions are then removed to reveal two portions of a rebus puzzle, which is a common phrase or name drawn out graphically. At this point the player may attempt to guess the solution to the rebus. The player maintains control of the board until the two selected game panels do not match, at which point control of the board is passed to the opponent. The game is over when one player correctly solves the rebus following making of a match during the turn. An example screenshot of a Concentration applet during a particular point in play is shown in Figure 9.

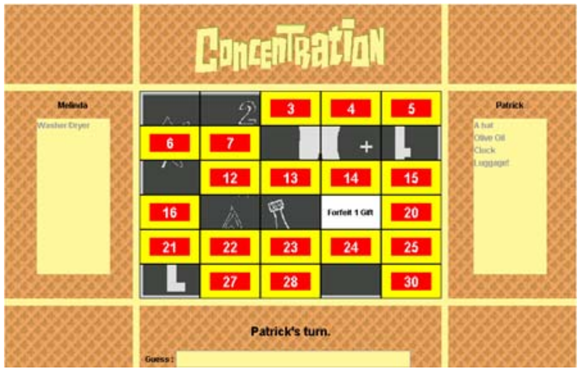


Figure 9. Screenshot of Concentration applet

As evidenced in Figure 9, the implementation of a game such as Concentration involves many components covered in the course. The rebus, a portion of which is visible in the figure, is an image that is painted onto the panel by a call in the paintComponent() method of the JPanel subclass to the drawImage() method of the Graphics class. The game panels consist of labels organized in a grid layout on an instantiation of a class derived from JPanel. Borders are used on each label to recreate the look and feel of the actual game. Each label is registered with a mouse listener so that, when selected via a mouse click, the prize associated with that game panel is displayed. After the second game panel is selected, if the prizes do not match, then a timer is actuated, with its action event handler used to redisplay the game panel's numbers. If the prizes do match, then the corresponding labels have their text and borders removed, are disabled (to turn off future mouse events), and made transparent to show that portion of the rebus that lies underneath. Furthermore, a text field (shown at the bottom of the figure) is made active for a limited amount of time so that, if desired, a guess as to the solution to the rebus can be entered. The prize won on that turn is placed into the current player's list of prizes, which is implemented using a text area. If a forfeit or take card is matched, then the appropriate player's text area is activated such that a prize can be selected for taking or for forfeiture. To insure that the game does not get stale when repeatedly played, random numbers are used to select one of several rebuses, and the prizes are placed into a list structure obtained from the collections framework, then shuffled so that with each game the prizes are placed into new positions on the board.

Results and Conclusions

Computer games are a powerful influence in our society. Many students who are now pursuing a career in the computer field first encountered computers as a gaming platform, and their interest in computers grew from that experience. Games can be successfully used as programming assignments for introducing graphical user interfaces, event handling, and other programming concepts. One of the strong points for this type of approach is that students have familiarity with a wide variety of games, and therefore have some reasonable expectations as to how the game is to operate. While many games are difficult to implement, a thoughtful approach can result in a positive learning outcome and a sense of accomplishment through the construction of appropriately selected games. Of particular interest are those games that interact with graphical elements; by using languages with built-in GUI capabilities along with IDEs that allow for ease of GUI editing, it is possible to tap into the interest that students have in playing such games. Student feedback obtained from course evaluation surveys has been strongly positive regarding this approach. Numerical data indicates that students feel that this type of course develops or enhances their ability to solve problems. Many have commented that they enjoyed the programming assignments because, once finished, the programs actually did something. Others commented that the assignments were found to be challenging, yet they were also interesting so they wanted to do it. The workload was considered to be greater but acceptable, as they often pushed themselves harder (especially when given opportunities for extra credit) and thereby learned more; some students have commented that they have learned more in this course than in any other course they have taken in college. The use of games made for a valuable learning experience in this course, both with the stated goal regarding GUIs and event handling, and through the introduction of other pertinent topics such as collection frameworks. The comment of one student sums it up best: the course was fun and educational.

Bibliography

1. J. Ross, "Guiding Students through Programming Puzzles: Value and Examples of Java Game Assignments," SIGCSE Bulletin, Vol. 34, No. 4 (December 2002), pp. 94-98.

2. K. Bruce, A. Danyluk, and T. Murtagh, "Event-driven Programming is Simple Enough for CS1," Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education, pp. 1-4, 2001, Canterbury, United Kingdom.

3. K. Walrath and M. Campione, The JFC Swing Tutorial: A Guide to Constructing GUIs, Addison-Wesley, 1999.

4. J. Estell, "The Card Game Assignment," Nifty Assignments 2004 Special Session, 35th SIGCSE Technical Symposium on Computer Science Education. Online: http://nifty.stanford.edu

5. Online: http://www.waste.org/~oxymoron/cards/

6. Card Games Rules Archive. Online: http://www.usplayingcard.com

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 interface design, programming applications, and ways to streamline the outcomes assessment process. Dr. Estell is a Senior Member of IEEE, and a member of ACM, ASEE, Tau Beta Pi, and Eta Kappa Nu.