REFACTORIZING THE CS1 COURSE*

Chuck Leska, John Rabung
Department of Computer Science
Randolph-Macon College
Ashland, VA 23005
(804) 752-3158, (804) 752-7328
{cleska,jrabung}@rmc.edu

ABSTRACT

Teaching object-oriented design and programming in CS1 can benefit from problems that stimulate student interest, that yield to natural analysis under an object-oriented approach, and that provide a basis for extension and reuse. Herein we describe our effort to design, develop, and implement a set of games intended to spawn such problems spanning nearly the entirety of concepts presented in our CS1 course. A primary target of our undertaking is to structure the games so that their development can be partitioned into sequential components suitable for use in lectures, labs, and student programming projects. The consequent course presentation focuses on problem solving, object-oriented concepts and design, and testing and handles syntax on a need-to-know basis within the context of the problem.

1. INTRODUCTION.

The nature of an academic setting with its all-too-short semesters and its consequently short project durations—made even shorter by efforts to accomplish multiple projects in a term—works in opposition to any effort to impress students with the need to design programs with software engineering principles in mind. Those projects, after all, are necessarily small and may never be revisited. Why bother with ideas of code reuse and maintainability? Herein we describe our attempt to develop and present a CS1 course that introduces an evolving project (or two) intended not only to permeate lecture, lab, and student projects throughout a semester, but which offers the prospect of reaching well into CS2 and even into a software engineering course beyond CS2. Under this approach, we feel that students will appreciate time and again the value of thoughtful design.
A number of issues and events led to the decision to develop our two "evolving projects" or "case studies" as the focus of our course. Our experience in the two previous offerings of a Java-based, objects-first CS1 has shown that applying the object-oriented paradigm to problems is not natural for our students—at least, not to the types of problems we have used typically in our CS1 courses. So, one priority of our course redevelopment has been to identify a set of problems that are appealing to our students and permit us to focus on problem solving, object-oriented concepts and design, and testing rather than focusing on the syntax of the implementation language or on problems whose orientation exercises the syntax du jour in designing solutions. Our goal is to cover syntax only as required by the shape of the solution of the problems we encounter.

This brought us to search for a textbook and an environment that supports such an approach and that provides visual feedback to assist students in understanding the change of state that occurs within objects when we instantiate or update them via a method call. The answer came in a single package with the selection of the text *Objects First With Java* by Barnes and Kölling and the *BlueJ* environment. The *BlueJ* object workbench and the close use of it by Barnes and Kölling in their many examples and interwoven exercises enables quick testing of developing classes, a point-and-click setting that mimics Java programming actions, and a freedom to develop syntax only when its time arrives.

Our previous CS1 experience has taught us that graphical output and feedback appeals to students. Having successfully introduced GUI development and event programming within the closed lab [2], we found that a gradual introduction to the necessary Java language elements with the students reusing the lessons of previous labs in subsequent exercises fosters a sense of mastery. The success of students on this material on the final exam encourages our reuse of this kind of activity. Thus, for our new offering we have sought problems where a GUI is a natural component or a useful extension, and, extending this notion slightly, where some very elementary graphical elements would enhance a project's appeal.

A secondary, but still important, consideration has come to us by way of recurring student comments in our Software Engineering course. These students observe that, although they believe they understand that applying software engineering principles and object-oriented design techniques leads to class and subsystem independence, they never experience the rewards of developing software in that manner. To the contrary, they find that the code they see and develop in this fashion yields systems that, to their eyes, are more complex and less intuitive than systems that they develop without using these techniques and design patterns. Although we do not expect to address this concern substantially in CS1, through practice and examples we do hope to lay the groundwork for making these principles more compelling in later courses. In particular, we plan to develop extensions to our original projects and new projects for use in subsequent courses (CS2 and Software Engineering) to let students experience first-hand the dividends of this approach.

We note that there are a number of papers in the literature that promote the use of case studies in CS1. For example, a paper by Nevison and Wells [4] both promotes this approach and provides a list of references to some of this literature. Aside from our goals, there are additional features that distinguish our project from these papers, and
particularly from [4]. For instance, although we utilize a small number of design patterns in our project design, MVC in particular, we do not believe that the level of experience and maturity of our students makes pursuing a serious discussion of this topic very promising in CS1. We intend to pick up the discussion of this topic later in our curriculum when students are better prepared to recognize the patterns in practice.

2. PROJECT DESCRIPTIONS.

During the summer of 2003 we undertook, as part of a supported project, the design and implementation of games to achieve the goals we detail in the introduction. We chose "MacMan", a variant of a project found in Mercer [3] (see section 7) and a pair of card games, the "Game of 25" and "War". Both card games are played with the standard deck of 52 cards. In this paper we limit our card game discussion to the Game of 25 and its precursors.

We have developed each project in a sequence of three versions tailored to the order of presentation to students. The design and code of these projects supports nearly the entire spectrum of design and programming topics that we cover in the CS1 course. In our development we use the MVC pattern in each version of each project.

The initial version of each is text based and provides the essential platform for the following versions. In particular, the development process involves adding a number of container structures as well as aspects of inheritance. We use three container structures: the ArrayList collection class, an array, and a two-dimensional array.

The second version of each project adds a GUI that initiates the actions of the game. There is a button for each action the player of the game can take. The construction is simple with the JFrame containing one JButton for each player action within a GridLayout. Each JButton click is an event in the ActionListener. The ActionListener responds by making a call to the appropriate operation in the Game class. Students develop all the requisite skills to create a GUI class through closed lab exercises.

The third version adds a graphics component. The nature of the component is quite different in each project. As in the second version, when students are asked to add the graphics to their project, the skills needed are developed within the closed lab.

2.1 MacMan

The play of the MacMan game is on a two-dimensional grid. A GridObject (a crumb, a cookie, a wall or the MacMan) occupies each position in the grid. The MacMan tours the grid for a fixed number of moves accumulating points by devouring crumbs and cookies. If the MacMan collides with a wall we decrement the number of moves remaining by a fixed amount. If the MacMan attempts to move off the grid, it dies and the game is over. The goal is to score as many points as possible.
2.1.1 Container classes

The array appears in MacMan to maintain the set of icons for display on the console. The grid setting for Macman is implemented with a two-dimensional array capable of holding a GridObject in each location.

2.1.2 Abstraction

In Macman, a GridObject offers the abstraction for any element that sits on the playing grid. We implement it as an abstract class. Rewards (crumbs and cookies), the MacMan and Blocks (walls) are subclasses of GridObject. The abstraction permits us to reuse the Grid and Viewer in each of the subsequent versions of the game.

2.1.3 Viewers and Control

In the first version of MacMan, the Viewer uses standard output to produce the view.

```
Current Score: 17
...
       .u.. 
#...#... 
..#....#.. 
..0...... 
...###... 
....0..0...
........# 
........0.. 
.##..#.. 
```

Figure 1: MacMan Console Viewer

In the second version of MacMan, the buttons in the control panel direct all the game and mover actions.

Figure 2: GUI Control for Version 2 of MacMan
For version three of MacMan, the GraphicsViewer displays two panels, a message panel and a paint panel. In the paint panel we paint each GridObject onto a square that corresponds to its position in the grid. We display the crumb as a small circle (FillOval()), the MacMan as a circle where we remove a slice to indicate its current direction (FillArc()), the cookie as two concentric circles, and the wall as a reddish square with 4 lines (drawLine()) to look like it's made from bricks. With very minor variations the grid elements in the graphic presentation are as found in Mercer [3].

![Figure 3: Graphics View for Version 3 of MacMan](image)

2.2 The Game of 25

The Game of 25 has a player and a dealer, each receiving 3 cards initially. Each card has a game value: face cards count ten points, aces count one or eleven points (whichever provides the optimal hand value), and numeric cards carry the number of points suggested by their face value. The object of the game is to have the highest total-hand value not exceeding 25 points. Player and dealer can choose to discard one card and receive one card from the deck to replace it. The player acts first with the dealer witnessing the outcome. The dealer then acts, but the player does not see the dealer's hand until the dealer's play is complete.

2.2.1 Container classes

In the Game of 25 we use an ArrayList to implement the Deck and CardCollection classes. An array appears in the card game to provide suit and face values for quick generation of a deck.

2.2.2 Abstraction

Inheritance occurs in the Game of 25 when we decide that the Game25Card class with its particular point scheme should be a subclass of a more generic Card class. This permits the CardCollection (it represents a hand in this game) and Deck classes to use the superclass Card and be reusable in implementations of different card games. The Viewer
class remains reusable since it utilizes only Card attributes to provide its display. It is only in putting cards in or getting cards from the Deck that we need to create from or cast to the Game25Card class.

2.2.3 Viewers and Control

In version one, the Viewer uses standard output to produce the game view.

```
Original Hands
---------------
Player Hand
10C 10S 8S
Dealer Hand
## ## ##

After Player Completes Play
----------------------------
Player Hand
10S 8S KS
Dealer Hand
## ## ##

Final Results
-------------
Player busts, dealer wins
10S 8S KS
Dealer Hand
7H 8H AC
```

**Figure 4: Game of 25 Console Viewer**

In version two, the game view remains the same but we add a control element to select the player action.

**Figure 5: GUI Control for Version 2 of Game of 25**

In version three, the GraphicsViewer displays three panels: a message panel and card panels for the player and the dealer. The cards are gif images (see section 7) that we place on JButtons. We name the image files so that it is easy to create paths to them from Card instance variables. This only requires the class Icon and the method ImageIcon(). All of this, then, is a straightforward extension of the GUI development work done in preparation for version two and an additional component of a lab where we introduce the new required Java elements.
3. STUDENT DESIGN IN CS 1.

Our expectations of student design in CS1 are not overly demanding. The substance of examples presented in class as well as that of the many examples in the textbook increases in complexity over the course of the semester, but the resulting designs still consist of a modest number of classes. The rise in complexity comes primarily from requiring classes to provide more interesting services. Our project assignments follow the same approach with the number of classes rising slowly as the semester progresses. We endeavor to be very clear about the rationale for adding a new class or refactoring an existing one, our discussion being bolstered by the textbook’s intuitive discussion of the use of cohesion and coupling in design. In class and lab students have ample opportunities to work on their design skills.

Our design goal for the semester is that our students become capable of crafting projects with overall design sophistication at about the level described in Section 2. We note that the quality of this target level is good for a CS1 course, but it is improvable. Rather than striving for the ultimate, we want the design to be accessible to first-year students in terms of its readability and its use of Java. One simple instance of where the target design could improve is in an application of sorting. The students write methods to separate hands of cards into suits and to order the cards within a suit by game value and house them in a single class. In CS2, we would expect the students to create separate Comparator and Sorting classes. In a sophomore-level software engineering class, we would discuss creating an abstract SortAlgorithm class and using a factory design pattern.

In our early student project assignments, we emphasize solely the game at hand – a specific card game – providing guidance for the design in the problem statement. As we approach concepts such as coupling and cohesion, inheritance and abstract classes, we discuss issues involved in lifting our projects to a more general level and how these concepts become fundamental to the design and generality of the project. For example, in our initial version of the Deck class, the Deck constructor initializes the cards it uses for the game. Once we discuss inheritance and the students are ready to think about the static and dynamic type of an object, the Game class creates the game cards and places
them in the deck with the type Card making the Deck independent of the particular card we use in the game.

4. USING THE CARD FRAMEWORK.

Because the card-game framework provides the structure that best suits the textbook's flow of topic presentation, we use it as the primary project focus for the course. In this section we provide a taste of how we weave the framework into the fabric of the course. We outline the details of the student projects and cite a couple of examples of the use of the material in class and in lab.

We begin our discussion of the basic Card class as a complement to the consideration of constructors, accessors and mutators prompted by the introduction of the TicketMachine class in Chapter 2 of the textbook. This leads quickly to the assignment of the first programming project in the second week of the course.

In this first "card game" each player receives just one card. (Card "dealing" is done at this early stage by instantiating a couple of Card objects on the BlueJ workbench and providing them as arguments to the method that plays the game.) The winner of the game is the player with the highest-valued card. If the players have cards with the same value, the game is a draw. The program is to display each player's card (as a string such as "5S" and "KD") and the result of the game. This project accomplishes a first experience with data class definition, constructors, object instantiation, selection, and parameters.

In the problem description of this first assignment we mention the MVC pattern. We do so to motivate the existence of the three distinct classes for the project. The classes are Card (the model of our data), Viewer (the view of the game data and result) and Game (the controller for the action of the game). A Card has three attributes of interest, a suit, a face value and a game value. The game value is not an element of our in-class design of the card, but students add it because the game requires it. Eventually we refactor the basic Card class and remove this attribute. The key methods to be developed here include the method that associates the game value of a Card object with its face value (the values range from 2 through 14 for, respectively, deuce through ace) and the method in the Game class that determines the results of a game.

In Chapter 4 of the textbook students encounter the ArrayList class. To supplement the book example and exercises we develop another trivial card game in class. In this game we deal two cards to each of two players. The player with the highest-valued card wins, or a draw is declared if both players hold cards of equal value. As a lab exercise we build a program for this game upon the classes implemented for programming project 1 and introduce a pre-written Deck class. After exercising and informally testing the methods of the Deck class, we develop a Hand class that acts as a motivator for use of the ArrayList class. Developing an algorithm for the playGame() method in the Game class, we then include in the Hand class only those methods we need to play the game. This exercise sets the students up for their second programming assignment at the end of the fourth week.

The goal of the third card game, the topic of the second programming assignment, is to have the hand with the largest sum of game values. There are still two players, but hands in this game are of arbitrary, pre-set size (set up front by the user). There is still no
strategy to implement—once a hand is dealt (this time dealt from an initialized and shuffled Deck object), a player evaluates the hand by summing its cards' game values. The program is to display each player's hand and the result of the game. This project exercises the development of a container class using an ArrayList and requires iteration.

As in the preceding lab exercise, we supply a Deck class for use in this assignment and treat it mostly as a black box. The model has two CardCollection classes to represent the two hands (the player's and the dealer's), a Card class and a Deck class. The CardCollection provides the usual operations for inserting, removing, or retrieving a card, for clearing the collection, and for determining the size of a collection or whether it is empty. The Game class serves dually in its controlling of the model and its acting as a part of the model. Its code to deal cards and to evaluate a hand does iteration via a while loop—the only iterative device the students know at this time. The public interface of the Game class consists of a constructor and a playGame() method. To start the game, a prompt (done using a JOptionPane feature) permits the player to enter the number of cards that will be dealt as a hand. The Viewer class implements methods to view a Card, a CardCollection and the Deck as well as the results of the game.

By the sixth week students are familiar with the Random class, with for loops and with a number of examples of using an Iterator within the ArrayList class. Both in class and in lab, students are working on problems requiring much more interesting logic. In week six, we assign the third project.

This assignment asks the students to implement a simplified version of the Game of 25. The game has a dealer and a player (the user). Both player and dealer receive 3 cards. As noted in Section 2 above, face cards here have a game value of ten and aces have a value of one or eleven, whichever provides the optimal score. The rules allow the player and dealer to optionally select one card to discard and replace it with one from the deck. The player goes first. A prompt allows the player to indicate whether he/she wishes to discard and, if so, which card in the hand to discard. The dealer's action occurs after the player has finished. The dealer takes action only if the player's hand is good enough to beat the dealer's current hand. The implementation of the dealer's strategy is left for the student programmer to decide; the program user decides the strategy of a player. In its presentation of a more complex setting with more involved logic and in its allowance of more design freedom, this assignment challenges students to think seriously about method cohesion, the use of private methods, the choice of class in which a method should be housed, and the logical structuring of conditional settings.

Within this assignment the Viewer implements some new responsibilities. After the initial deal, the program is to openly display the player's cards and display the dealer cards "face down"—as the string "##". The implementation we prefer (but not all students take up) adds an attribute to the Card class to signal whether we present the card face up or face down. When the dealer action is complete, the user is to see a display of both hands with all cards face up along with the result message. Once again the Game class provides the control for the game.

During weeks 7 and 8 we introduce arrays and begin the discussion of sorting. In the card framework students work in lab on methods that gather cards into suits and place the cards in game value order within a suit. In lab the students learn about developing a simple GUI with buttons with action listeners that call to execution the basic actions of
a Tic Tac Toe example. By week 9 we have begun discussing inheritance. As their fourth programming assignment students are to set up a GUI controller for the Game of 25 and factor the former Card class into a generic Card class and a Game25Card subclass in anticipation of implementing a different card game in the next assignment. In addition, they are challenged to set up a graphic display of cards in the hands as icons on a bank of buttons. Depending on the student's foresight in implementing previous projects, introducing the GUI and allowing the user to play the game multiple times may require more or less refactoring.

Weeks 10 and 11 bring explicit discussions and examples of refactoring to promote coupling and cohesion. These weeks include a lab "design workshop" where students work in teams to develop a design for the MacMan project. In the following lab they implement this project along with a GUI and the aforementioned graphic element. Their final programming project has them attempting to reuse and extend (to appropriate subclasses) the generic portion of their card game code in developing a different simple game, "War", that requires recognizing a new set of game values for cards and establishing new uses for the CardCollection class.

In the final weeks of the semester we consolidate what we have learned through the development and refinement of projects, particularly the card game and the MacMan game. In doing so, we made improvements as needed and look ahead to what else might be done, giving a connection to the next course, and perhaps beyond.

5. WHAT HAVE WE LEARNED?

We have found this to be a most stimulating experience. Unencumbered by the pressure to build up a complicated foundation of just-do-it-like-this-now-we'll-talk-about-it-later features of Java (e.g., file I/O or even a header like public static void main() ) and freed of any need for a student-assistance package of classes that sidestep such features, we enjoyed simply teaching programming concepts and introducing features as we needed them. We found ourselves not dwelling so much on syntactic detail, but instead caring very much about solving problems at hand and making sensible, forward-looking design decisions.

Class topics and discussions seem to flow more naturally. Rather than the mindset that "today we have to talk about for loops", ours has been "today we'll need for loops to get where we want to be in this project". Because our approach to student development of concepts and skills is so context sensitive, there has been a natural and welcome merging of lecture, lab, and projects so that the lines between these have become blurred. We find ourselves on occasion quitting the lecture room to go into a lab session or injecting a lecture segment into a lab exercise. In effect, this approach has led us naturally into a more active teaching/learning style.

But this is not without its drawbacks. For one, it depends on the student assuming a greater sense of responsibility for his/her progress in the course. Although the increased activity in lab/lecture helps to heighten this sense, presuming that students are reading the text and keeping apace with the exercises conveniently positioned throughout the textbook is still a mistake. We need to more strongly encourage/enforce such reading lest students wrongly suppose that they can ride the coattails of their lab partners through this
course. Despite trying to demonstrate the value of active, careful reading early in the
course – we make the first lab a walkthrough of the reading and exercises in the first
chapter of the textbook – it is now clear that some of our students do not get the message.

Perhaps with a shifting away from a teach-the-language approach toward a
use-the-language approach we should have expected perplexing new problems. One
glaring such problem is that it seems harder to sense what students are learning. In one
week's laboratory the whole class might work quickly and easily through a set of
exercises and then show utter confusion in the next week's lab. This is due, no doubt, to
our inability to predict which concepts have taken hold and which have not in the interval
between labs. In demonstrating the sometimes-wide gap between discussing concepts and
applying them, this underscores the need for students to stay actively involved outside of
lecture and lab. Lacking some of this active student involvement and lacking our ability
to closely gauge student progress in a problem-oriented environment, we sometimes draft
overly ambitious lab exercises. Nonetheless, even utter confusion seems to motivate
fruitful discussion topics for following class periods and tends to heighten student
attentiveness.

What can we take from this CS1 course into CS2? Certainly we can use our CS1
projects as examples and as platforms for further development. A card game platform
offers opportunities for devising new games (or citing old games) that can make creative
use of some of the basic structures generally covered in CS2. It is not hard, for example,
to imagine adding to a card-game project container classes such as stacks (discard piles
from which players may redraw) or queues (discard piles which are turned over and
reused at certain times during play) or sets (does a card collection contain some card of
a given face value or suit) or maps (given a card, as key, return its image, as value).

Likewise we can build on the program-design concepts coming out of CS1. Students
have experience with refactoring, with reuse, with some consideration of coupling;
they've seen the MVC pattern being used in a number of settings and developed their own
container classes. CS2 will do more of this kind of work, but will benefit from having
students who know up front some of the value of the work.

What extra burdens do we bear in CS2 from having taught CS1 as we did? Certainly
some of what might be considered "usual" content of CS1 was not covered – for instance,
we did no file I/O, no exception handling, and never even saw a try – catch block. These
topics will have to be picked up in CS2, but, we think, our now more mature student will
receive these topics more efficiently.

One final note – we could not have undertaken this project without an environment
like BlueJ with its object workbench that allows easy point-and-click instantiation of any
class and easy testing of the behavior of these instantiated objects. And the
Barnes-Kölling textbook has been invaluable in its order and style of presentation.
Perhaps we should make the textbook authors and the BlueJ designers co-authors of this
paper since their works have been so enabling to us in taking our new approach.

6. ACKNOWLEDGMENTS.

When looking for texts for the course we happened upon Rick Mercer's Grid
example [3]. It inspired our MacMan project and the images provided by the Viewer
class. For allowing us the use of the card images that accompany their textbook [1], we gratefully acknowledge Jim Cohoon and Jack Davidson. We thank the students who are sharing this work with us by participating in our CS1 course as well as those in more advanced courses who designed and implemented some of our project extensions and critiqued aspects of our design and implementation. We are grateful to the Walter Williams Craigie Teaching Endowment for its partial support of this project.

7. REFERENCES.

2. Leska, C. J., "Learning to Develop GUIs in Java Using Closed Labs", *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE 2003), 228.
8. APPENDIX - CLASSES FOR PROJECTS
MacMan Project

Figure 7: UML for MacMan

Game of 25 Project

Figure 8: UML for Game of 25