The Game of Set® – An Ideal Example for Introducing Polymorphism and Design Patterns

Stuart Hansen
Computer Science Department
University of Wisconsin — Parkside
Kenosha, WI 53141
hansen@cs.uwp.edu

ABSTRACT

This paper presents an object–oriented design for a solitaire version of the game of Set[1]. The design is responsibility driven and illustrates polymorphism and several fundamental design patterns, including Flyweight, Strategy and Factory. It introduces each of these to solve particular problems within the design. The direct application of these concepts and the interest our students show in the game make Set an ideal example for classroom discussions and assignments.

Categories and Subject Descriptors

D.1.5 [**Software**]: Programming Techniques—*Object–Oriented Programming*; J.m [**Computer Applications**]: Miscellaneous

General Terms

Design, Languages

Keywords

Design Patterns

1. INTRODUCTION

Polymorphism and design patterns are frequently treated as advanced topics in object–oriented programming. They are introduced in upper division courses when students start dealing with issues of large scale development. Efforts are being made to introduce these ideas earlier in the curriculum [1, 5, 6, 7, 8, 9], but broad acceptance of their early introduction has not been achieved. This is partly because it is difficult to find sample problems that are approachable early in the curriculum and whose solutions use polymorphism and design patterns in simple direct ways.

The game of Set is an ideal example for introducing these concepts. An object–oriented design of the game uses polymorphism and design patterns in ways that are easy for our students to grasp.

[1]Set is a Registered Trademark of Set Enterprises Inc.

SIGCSE'04, March 3–7, 2004, Norfolk, Virginia, USA.

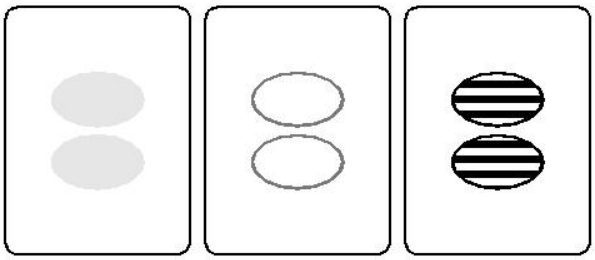

Figure 1: The cards in this figure form a set. They all contain two ovals so they agree on the SYMBOL and NUMBER properties. Each has a different SHADING and each has a different COLOR (here represented by varying shades of gray).

One of the strengths of this example is that it neatly shows the transition from simpler, already understood ideas like class–instance, to using the more complex notions of polymorphism and design patterns. Our discussion in this paper follows the same flow of the ideas as presented during our class discussions.

2. THE GAME OF SET

Set is a simple card game. The object of the game is to identify a "Set" of three cards from 12 cards laid out on the table. Each card has a variation of the following four features:

(A) COLOR: Each card is *red*, *green*, or *purple*.
(B) SYMBOL: Each card contains *ovals*, *squiggles*, or *diamonds*.
(C) NUMBER: Each card has *one*, *two*, or *three* symbols.
(D) SHADING: Each card's symbols are *solid*, *open*, or *striped*.

A "Set" consists of three cards in which each feature is EITHER the same on each card OR is different on each card.[2] When a set is found, those cards are removed and replaced by three new cards from the deck. The deck is made up of 81 unique cards, containing all the possible combinations of the four features. The game continues until the deck is exhausted.

Figures 1 and 2 show sets. In Figure 1 all three cards contain two ovals, so the set property is met for the NUMBER and SYMBOL features. The first is solid, the second open and the third striped, so they disagree on SHADING. The figure uses different shades of gray to represent colors. The three cards disagree on COLOR. Since the cards agree or disagree on each feature, they form a set. In Figure 2 the cards agree on COLOR and SHADING. They disagree on SYMBOL and NUMBER.

Figure 3 shows three cards that do not form a set. There are

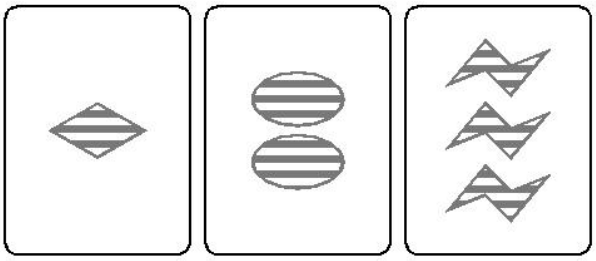

Figure 2: The three cards in this figure form a set. Each one contains a different SYMBOL and a different NUMBER of symbols. They are all striped and they are all the same COLOR.

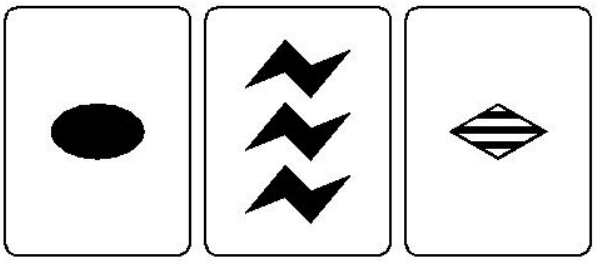

Figure 3: The outside two cards each contain one symbol, while the middle card contains three symbols. The set property is violated for NUMBER. The other features do not matter. The cards do not form a set.

two cards with one symbol and one card with three symbols. The NUMBER feature is neither different on all three cards nor the same on all three, so they do not form a set.

We have developed an object–oriented design of Set as an example in several sections of an introductory Java course. The students in this course have programming experience in other languages. Their software development skills are typically similar to those of advanced CS2 students.

Our design is for a solitaire version of Set. Figure 4 shows an implementation of the design during a run. Twelve cards are laid out in a 3x4 grid. The user identifies sets by clicking on the cards.

Set is an intriguing game to the students. Several of our students have purchased decks of Set cards following our class discussions. The author has visited the student lab several times when games were in progress. While Set may be played competitively, winning and losing seem secondary to the simple pleasure of finding sets. The students' fascination with the game helps motivate and energize the class discussions.

3. DESIGNING THE SET CARDS

A solid design for the individual cards is imperative if the overall application is to work well. There are only two operations in which the cards participate. Cards are displayed on the screen and collections of three cards are checked to determine if they form a set. The goal of this section is to develop a card design that cleanly implements these operations.

The card's features are constructed when the deck is created. The features do not change during the run of the program. Figure 5

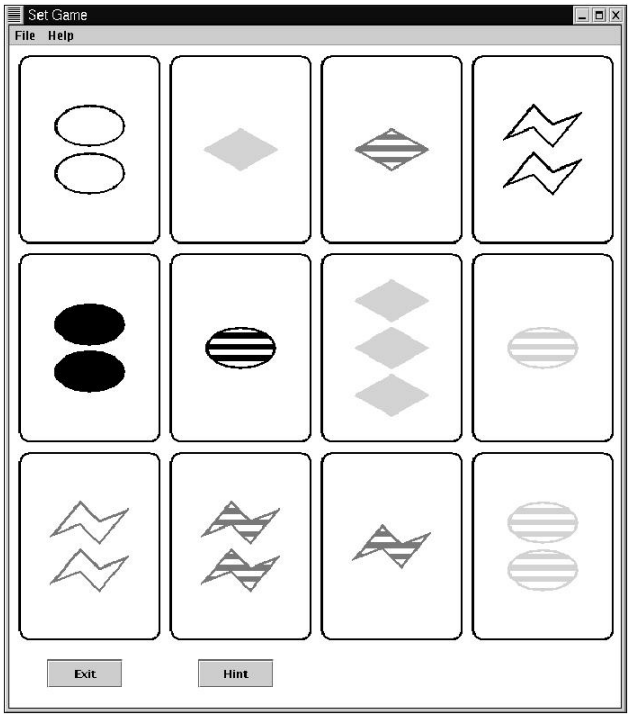

Figure 4: This figure shows the top–level GUI of our Set program. The game is played by displaying 12 cards and searching for sets. When a set is found, those cards are replaced by three new cards. In this figure, numbering the cards from 1–12 in row major order, cards 1, 7 and 11 form a set, as do cards 6, 7 and 9.

shows a high level class diagram for Set cards. Each of the card's four features becomes an attribute of the Card class. A few of our better students immediately query why two of the features are represented by classes and the other two are represented by interfaces. This question is central to the introduction of polymorphism and we postpone its discussion until we have implemented the two simpler features: COLOR and NUMBER.

3.1 Comparing Cards

Determining if three cards form a set requires their features to be checked for equality. The features are represented by objects, so we must guarantee that object equality means the same thing as feature equality. For example, two cards containing squiggles should have their `symbol` attributes equal, since the SYMBOL feature is the same on both. Similarly, two striped cards should have equal `shading` even if one fills symbols with green stripes and the other fills symbols with purple stripes.

The solution to this problem usually suggested by our students is to write an `equals()` method for each attribute. Java's AWT Color class already contains an `equals()` method. We can use this method to check for `color` equality. Each of the other attributes would require a similar `equals()` method. This solution works, but requires tedious programming.

3.1.1 The Flyweight Design Pattern

A much simpler solution is to use the Flyweight Design Pattern[3]. The flyweight pattern uses shared attribute objects to support multiple objects having the same attribute values. The flyweight pattern can be used for each of the features. For example,

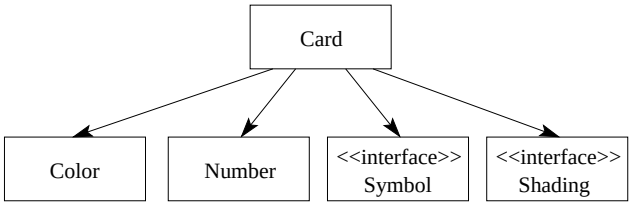

Figure 5: A card contains four features that uniquely identify it in the deck. The features are used for displaying the cards and determining if three cards form a set.

we will use the same squiggle object for every card that contains squiggles. The squiggle symbol is the same on a card that has three green striped squiggles as on a card that has one red solid squiggle, so sharing the object should not pose any problems.

The flyweight pattern limits the number of objects needed for the attributes. There are 81 cards, each with four features. If each card has a unique copy of its attributes the implementation will have $81 * 4 = 324$ attribute objects. Each feature has three possible values. Using the flyweight pattern there will be only $4 * 3 = 12$ attribute objects shared among all the cards.

In our design, the major advantage of using the flyweight pattern is that it lets us compare attributes directly. In our Java implementation we can compare colors, numbers, symbols and shadings using `==` and `!=`. These work because an object always `==` itself. There is only one instance of the squiggle object shared among the cards, so two cards that are both squiggles must share that instance and `card1.symbol == card2.symbol`.

3.1.2 Checking for a Set

Now that we have an efficient way to compare attributes, the game can determine if three cards form a set. The conditions needed for this are quite complex. Java–like pseudocode for a method to check if the `symbol` attribute satisfies the set property is:

```
boolean isSetSymbol (Card card1, Card card2,
                                    Card card3)
{
    if ((card1.symbol == card2.symbol
                && card2.symbol == card3.symbol)
       || (card1.symbol != card2.symbol
                && card2.symbol != card3.symbol
                && card1.symbol != card3.symbol))
       return true
    else
       return false
}
```

Each of the other features requires a similar method. Three cards form a set if and only if all four `isSet...()` methods return true.

3.2 Displaying the Cards

Getting students to think in terms of assigning responsibilities to objects is one of the major challenges we face in teaching object–oriented design [4, 10]. Displaying Set cards provides an excellent example of distributing responsibilities to the appropriate object. Cards are responsible for knowing how to draw themselves, but depend on their attributes to accomplish this. Some attributes passively provide data, while others take active responsibility for some aspect of the drawing.

3.2.1 Color

Every card is assigned a color. The card's color is used to both draw the symbols and, for solid and striped cards, to assist in filling the symbols. Java's AWT Color class provides all the functionality needed. When drawing the symbols, the color is treated as a passive property which is passed to the graphics environment. The responsibilities of color when filling symbols is more complex and is discussed below when considering the card's shading.

3.2.2 Number/Locations

The card's NUMBER tells how many SYMBOLs are on the card. When implementing a card, however, there is the additional responsibility of knowing where each of the symbols is to be drawn. It is natural to assign this responsibility to the `number` attribute, as well. All cards with three symbols will draw those symbols at the same relative locations. `number` is no longer a simple integer, but is implemented as a vector of (x,y) locations. In fact, in our implementation, the attribute is named `locations`. Each location represents a point where a symbol is to be drawn. The NUMBER feature is now represented by the size of the `locations` vector.

3.2.3 Symbol

`Color` and `locations` provide data for the card to use when drawing itself. The `symbol` provides a shape. When first discussing the design of Set cards, students often suggest representing the symbol with a string. To draw the symbol, the card would examine the string and invoke the appropriate code. Responsibility driven design suggests, instead, that a `symbol` should be responsible for knowing how to draw itself. To accomplish this, the `symbol` provides a `draw()` method that renders its shape onto the card.

3.2.4 Polymorphism

The symbols provide a concrete, easy to understand application of polymorphism. When treated from a responsibility point of view, extending the concept of an attribute having responsibility for data to having responsibility for a method is a simple leap for our students.

The cards' symbols are best implemented using multiple classes. Since all objects of the same class have identical methods, each `symbol` type must belong to its own class to have a different `draw()` method. Figure 6 shows the class hierarchy for symbols. `Symbol` is an interface that declares the `draw()` method. The `Symbol` interface is implemented by three different classes, `Diamond`, `Oval` and `Squiggle`. Each class contains a `draw()` method that knows how to draw the appropriate shape.

3.2.5 The Strategy Design Pattern

We can bring `color`, `locations` and `symbol` together to see how the attributes collaborate to draw a card. The card's `draw()` method uses `color` and `locations` to guide the drawing. It delegates the responsibility for drawing the shapes to `symbol`. The notion of an algorithm delegating responsibility for some part of its function to an appropriate helper object is called the Strategy Design Pattern [3]. Utilizing the strategy pattern to draw cards simplifies the card's `draw()` code. The trade offs are that we must define the `Symbol` interface and that each symbol class is required to contain a `draw()` method.

Java–like pseudocode for the card's draw method is:

```
void draw () {
    setColor(color)
    for each location {
        symbol.draw (shading)
    }
}
```

3.2.6 Shading

The design of the SHADING feature is conceptually the most complex. Fortunately, most of our students have experience with paint programs. They already understand that filling may be done with a solid color or with a texture, a periodic pattern of multiple colors. Concepts are clarified by making the correspondence explicit between the drawing needed by the Set program and the drawing they have previously done.

Java provides a `Paint` interface that specifies how a region is to be filled. Classes implementing the `Paint` interface must supply a `paint()` method. Java's libraries include several classes that implement the interface, including `Color` and `TexturePaint`. Figure 7 shows the class hierarchy used by the Set program for filling symbols. The paint objects require no new class definitions. They are all instantiations of predefined Java classes.

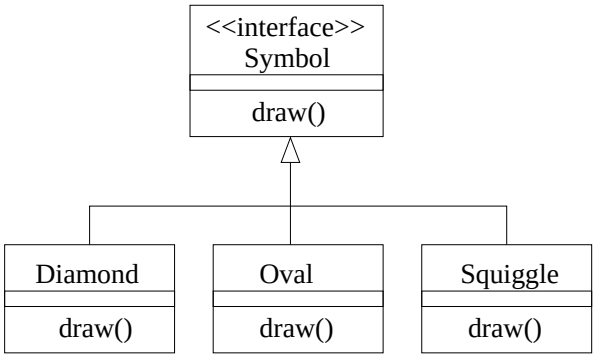

Figure 6: The `symbol` attribute determines whether a card contains diamonds, ovals or squiggles. The `Symbol` interface requires that each implementing class contain a `draw()` method. This method differs in each implementation, drawing the appropriate shape for that class.

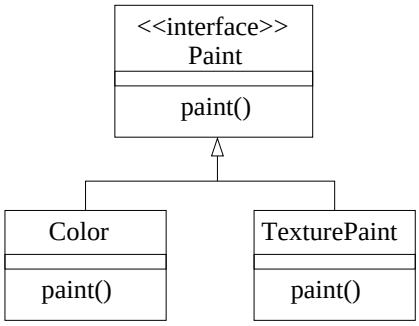

Figure 7: The `Paint` interface and its two implementing classes are part of Java's AWT library. Solid and Open cards use `Color`s for painting. Striped cards use a `TexturePaint` object.

A solid card uses its `color` attribute for painting. Up to this point, most students have thought of colors only as passive data objects. Now, the card's `color` takes on an active responsibility for filling regions. Thinking of colors as active objects, participating in the drawing of the cards, is a natural result of the responsibility driven approach to design.

An open card uses the background color (off white) for painting. Striped cards require an instance of `TexturePaint`. The `TexturePaint` object is constructed using both the card's color and the background color to create the texture's pattern.

3.2.7 The Factory Design Pattern

The card's `shading` attribute does not implement the `Paint` interface. If it did, the program could not compare shadings using `==`. For example, a solid purple card uses the color purple for painting. A solid red card uses the color red. They use different paint objects, but have the same `shading`, solid. Testing `shading` for equality should return true, but testing the `Paint` objects returns false.

The solution to this problem again lies in design patterns. The Factory Design Pattern has an object return an instance from a family of related classes that meet this case's particular needs [3]. The `shading` attributes act as `Paint` factories. Each `shading` object has a `getPaint()` method that returns a concrete instance of `Paint`. The `getPaint()` method uses the card's `color` and the background color to create and return a `Paint` object. If the `color` differs between two cards, different `Paint` objects are returned. Thus, all solid cards still share the solid shading attribute, but use different `paint()` methods for different colors. The additional level of indirection lets the program compare `shading` attributes and still paint cards appropriately. Figure 8 shows the class hierarchy for `Shading`.

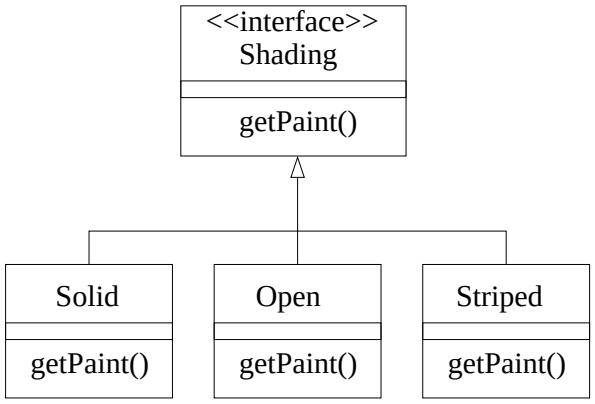

Figure 8: The way the symbol is filled is determined by the card's `shading`. The card's shading does not contain a `paint()` method, but rather is a paint factory. It has a `getPaint()` method that returns an appropriate `Paint` object. Separating the `Paint` classes from the `Shading` classes allows the programmer to easily compare the shadings.

4. PUTTING THE PIECES TOGETHER

Once the Set cards are designed, the remainder of the design falls into place relatively quickly. Students have little trouble designing the deck of cards or the application itself. The deck consists of an array of cards with methods to shuffle and deal one card. The application has a relatively simple graphical user interface with a few menu options and mouse events.

After developing the design during class, we assign students to implement some or all of it. While this example is not large, some of the ideas are new and challenging to the students. We believe students can understand and appreciate the design concepts without doing a complete implementation. We have had much success assigning partial implementations. Typically, we provide students with a shell program, including the interfaces and ask them to implement many of the concrete classes.

5. SUMMARY

The object-oriented design presented in this paper has emphasized responsibility driven design, polymorphism, and design patterns. We have found that introducing these concepts while solving specific problems is very worthwhile. It instills in students the power of the ideas and gives them a touchstone example for future reference. The Set program provides a particularly clear example of this instructional approach.

Java source code for the solitaire Set game as presented in this paper may be downloaded from:

`http://www.cs.uwp.edu/staff/hansen`

6. REFERENCES

[1] J. Adams and J. Frens. Object centered design for java: Teaching ood in cs–1. *Proceedings of the Thirty–Fourth SIGCSE Technical Symposium on Computer Science Education*, 34:273–277, 2003.

[2] M. Falco. Set: The Family Game of Visual Perception. `http://www.setgame.com`, 2002. URL current as of August, 2003.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1995.

[4] C. Larman. *Applying UML and Patterns: An Introduction to Object–Oriented Analysis and design and the Unified Process*. Prentice–Hall, 2 edition, 2002.

[5] D. Nguyen and S. Wong. Design patterns for decoupling data structures and algorithms. *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, 30:87–91, 1999.

[6] D. Nguyen and S. Wong. Design patterns for lazy evaluation. *Proceedings of the Thirty–First SIGCSE Technical Symposium on Computer Science Education*, pages 21–25, 2000.

[7] D. Nguyen and S. Wong. Design patterns for sorting. *Proceedings of the Thirty–Second SIGCSE Technical Symposium on Computer Science Education*, pages 263–267, 2001.

[8] D. Nguyen and S. Wong. Design patterns for games. *Proceedings of the Thirty–Third SIGCSE Technical Symposium on Computer Science Education*, pages 126–132, 2002.

[9] M. R. Wick. Kaleidoscope: Using design patterns in cs1. *Proceedings of the Thirty–Second SIGCSE Technical Symposium on Computer Science Education*, pages 258–262, 2001.

[10] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object–Oriented Software*. Prentice–Hall, 1990.