Software Reuse with Shuffler Design Pattern

G Priyalakshmi[1], R Nadarajan[2], S Anandhi[3]

Department of Applied Mathematics and Computational Sciences,
PSG College of Technology

`priya.venky2001@gmail.com, nadarajan_psg@yahoo.co.in, anandhigokul@gmail.com`

Abstract. Design patterns are used in software development to provide reusable solutions. Patterns can improve the documentation and maintenance of existing systems. Shuffle is a method which refers to jumble together, mix, or interchange the positions of objects. The shuffling technique is applied to gaming applications like Jigsaw puzzle, Sudoku, jumbled word game, etc. In music software, shuffling refers to the ability to randomize the order of the songs in a playlist. Thus, shuffling has a vast number of known uses in the software industry. The purpose of this paper is to contrive a design technique from existing solutions for shuffling, to make them more accessible to the developers of new systems. The Shuffler design pattern helps to choose generic design alternatives that make a system reusable. The Shuffler design pattern has many implementation variants; as a consequence it can be applied to many systems. The pattern has been documented in a consistent format. Concrete examples were given, because they help a software practitioner see the design in action.

Categories and Subject Descriptors

•Software and its engineering → Software creation and management → Software development process management → Software development methods → Design Patterns

•Software and its engineering → Software notations and tools → Language Features → Patterns

General Terms

Patterns, Design, Shuffling, Identity

Keywords

Shuffler, Design Patterns, Reusability

1. Introduction

Patterns have been introduced by Christopher Alexander in the field of architecture. Reusable architectural proposals for producing good quality designs were documented [8]. In the mid-90s, the idea of patterns was adopted by object oriented software developers. In 1995, the so-called GoF (Gang of Four, Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides), cataloged 23 design patterns aimed at meeting some commonly-recurring object-oriented design needs [1].

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder [1]. Design patterns make it easier to reuse successful designs and architectures. When designers find a good solution, they use it again and again. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help to choose design alternatives that make a system reusable and avoid alternatives that compromise reusability.

Design patterns vary in their granularity and level of abstraction. Based on the first criterion, patterns can have creational, structural, or behavioral purpose. The second criterion, called scope, specifies whether the pattern applies primarily to classes or to objects. The Shuffler pattern perceived in the paper falls under the category of Behavioral Object patterns. This category is perfect because of the reason, behavioral object patterns use object composition rather than inheritance.

The Shuffler design pattern was learnt from the existing solutions. Shuffling is most commonly used in a Music Player application, where the entire music library is shuffle played or a particular artist's songs are shuffle played. Another well known use of shuffling is in games. The famous Jigsaw Puzzle uses unassembled jigsaw pieces of different shapes or colors. Each time before a new game is commenced, the pieces are randomly shuffled. The third common game used widely is the Word Jumble. It is a word puzzle with a clue, a drawing illustrating the clue, and a set of words, each of which is "jumbled" by scrambling its letters. The letters must be a subset of the 26 English alphabets, which form the answer.

The pattern is described using graphical notations, which capture the end product of the design process as relationships between classes and objects. The pattern description also records the decisions, alternatives and trade-offs that led to the design. The template makes the design pattern easier to learn, compare, and use.

Every pattern is powerful enough to represent objects of different granularity, which may vary in size and number. Objects can signify any real world entity, which may be hardware, or an entire application. The Shuffler pattern represents everything down to the jigsaw pieces or all the way up to employees in an organization.

Apostolos et al. in their paper [2] have summarized the state of research related to design patterns existed up to now. This paper falls under the category of *"Miscellaneous Issues on Design Patterns"*, since it documents design decisions related to shuffling in real-time.

The pattern Shuffler is described in a template with the sections like, Pattern Name and Classification, Intent, Also known as, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample code, Known uses, and Related Patterns.

2. Pattern Description

The design pattern in this paper is a description of communicating objects and classes that are customized to solve a generic design problem in a particular context. Each pattern is divided into sections according to the following template. The template lends a consistent structure to the information, making design patterns easier to study, evaluate, and apply.

2.1 Pattern Name And Classification

Shuffler. We put this pattern in the category of Behavioral patterns. This pattern is in the Object scope [1].

2.2 Also Known As

Virtual Rearrange

2.3 Intent

Provide shuffling techniques in various forms. Shuffler lets the algorithm vary the objects to be shuffled independently from the clients that use it.

2.4 Motivation

Consider a jigsaw puzzle gaming application. Jigsaw puzzles come in 300-piece, 500-piece, 750-piece, etc, the largest has 32,256 pieces. There is also a cardboard jigsaw puzzle, a 3-D puzzle which is a puzzle globe. The computer versions of jigsaw puzzles have the advantages of requiring zero cleanups and no risk of losing any pieces. Many computer-based jigsaw puzzles do not allow pieces to be rotated, so all pieces are displayed in their correct orientation. These puzzles are thus considerably easier than the puzzles in which the pieces are thoroughly shuffled.

The problem here is random shuffling of the puzzle pieces in a short span of time. The users can choose their own puzzle size, cut design, and image, or upload their own images to be used as puzzles. Furthermore, it's clear that the shuffle request from the client may be handled by one of the several shuffler objects, which depends on the type of shuffling. They are either shuffling the whole object, or shuffling a part of the object, and selecting only few objects in the shuffling object list. The shuffling method like Fisher-Yates algorithm, or Sattolo's algorithm, can also be changed due to the abstraction provided in the design pattern.

The idea of this pattern is to create a collection of ShufflingObj objects as shown in Figure 1. The class 4PieceShuffler creates the list and initiates shuffling. If the client requests partial shuffling of objects, the identityPart remains the same, and the shufflingPart is swapped among the objects in the list.

The Strategy pattern merged with Composite pattern was also studied [10], but Composite pattern was not suitable to the problem the authors explored. The Shuffler design pattern is unlike Strategy design pattern in multiple ways. Strategy defines a family of algorithms and makes them interchangeable, but Shuffler interchanges objects in a collection. Hence both patterns can be merged to give a better solution. The client has the flexibility to change the algorithm at run-time in Strategy pattern, while in Shuffler, the client has the ability to change the shuffling algorithm at run-time.
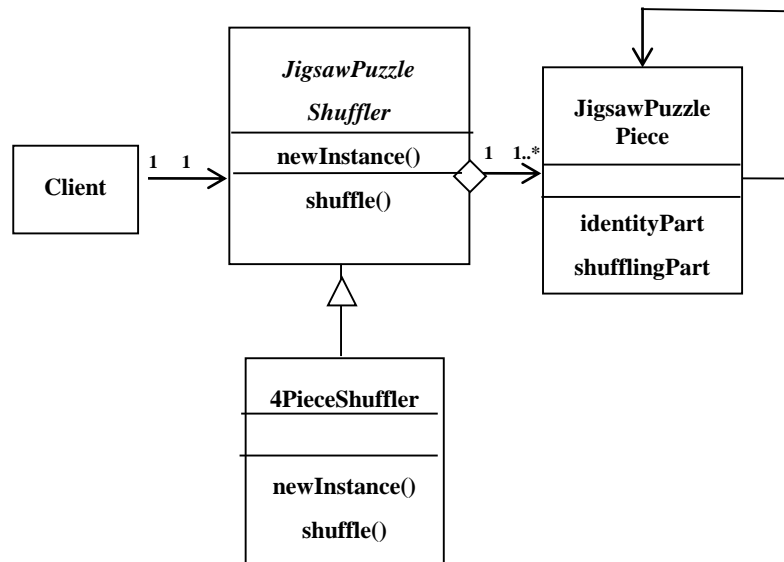
Figure 1. Design Class Diagram for Jigsaw Puzzle

2.5 Applicability

Use the Shuffler pattern in one of the following situations:

- When an abstraction has two aspects, one which will remain unaltered for a long period, and one which will undergo change often.

- When a collection of objects are instantiated from a same class, or classes derived from a common abstract class.

- The roles of the collection of objects can be swapped.

- A subset of objects can be considered for shuffling.

2.6 Structure

The structure of the design pattern Shuffler is depicted in Figure 2. The Client class will make a shuffle request initially to the one of the objects of the class ConcreteShuffler. The object of ConcreteShuffler class takes charge of the shuffling request. It also has the privilege to create a new object which is placed in the collection of objects of ShufflingObject. The AbstractShuffler gives an opportunity to the developer to implement various types of shufflers, depending on the system which is developed. The objects of ShufflingObject class are linked to each other in any convenient form as per the system requirements. When shuffled, few members are rearranged, while the other few are retained in their previous objects.

2.7 Participants

The classes and objects participating in this design pattern are:

- **Client -** Initiates the shuffle request to the ConcreteShuffler object.

- **Abstract Shuffler (JigsawPuzzleShuffler) -** Declares an interface for shuffling the composite object.

- **ConcreteShuffler (4PieceShuffler)** - Handles the shuffling request it is responsible for. Can create a new object which is inserted in the list of shuffling objects.

- **ShufflingObject (JigsawPuzzlePiece) -** Links to the next object in the list. Comprises of an identity part and shuffling part. When rearranging, the identity part remains the same, only the shuffling part will change.

2.8 Collaborations

The interactions between participant objects to carry out the responsibilities are listed below:

- When a client issues a request to rearrange the objects, the state which represents the identity part remains unaltered, but the state which represents the shuffling part gets swapped.

- The client issues a request to the ConcreteShuffler to add a new object to the collection.
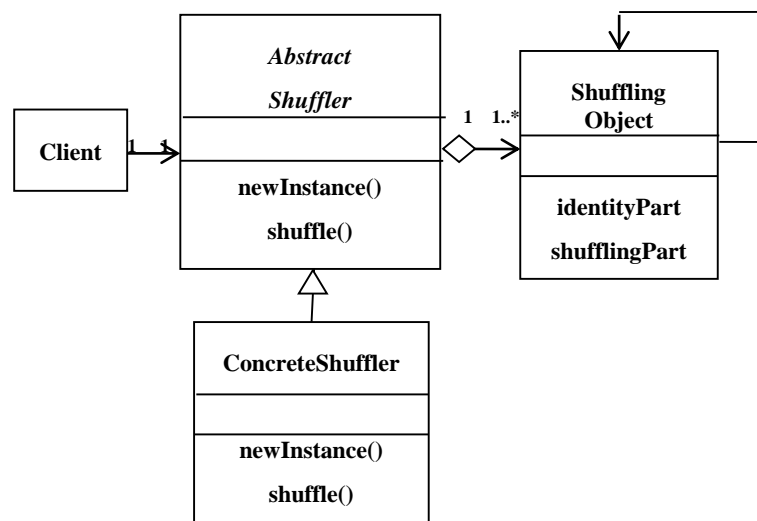

Figure 2. Structure of the Shuffler Design Pattern

The following interaction diagram shown in Figure 3, illustrates the collaborations between the ConcreteShuffler object and the collection of ShufflingObj objects.
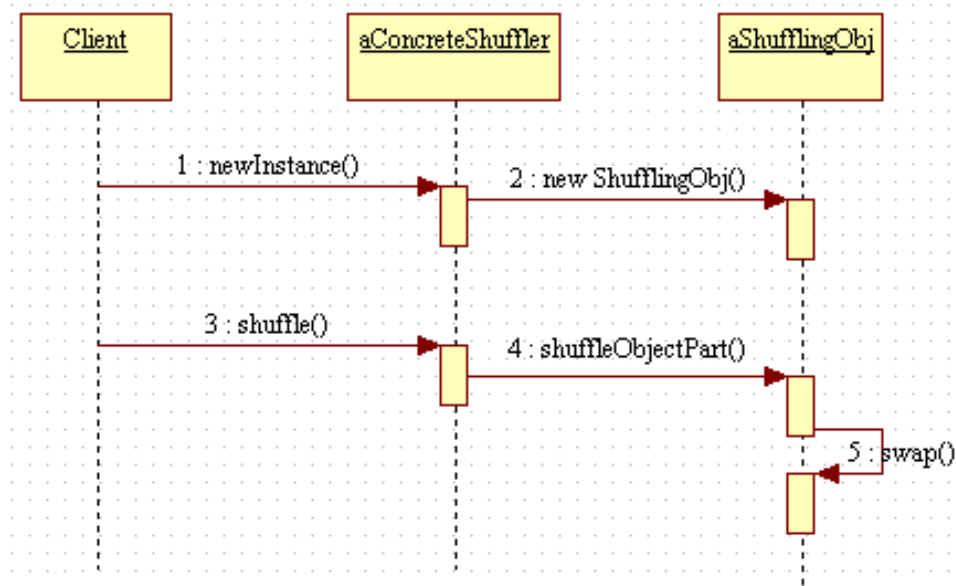

Figure 3. Interaction Diagram

2.9 Consequences

Shuffler pattern has the following benefits and liabilities:

1. **It supports shuffling by preserving the identity of the object.** Shuffling will be done in multiple ways. For example, the whole object will be shuffled as in DNA Shuffling [3], extending its range of applications to small molecule pharmaceuticals, gene therapy vehicles and transgenes, vaccines and evolved viruses for vaccines, and laboratory animal models. In a jigsaw puzzle, the pieces will be shuffled as a whole.

    Consider employees working for a company with individual roles. After a period of time, assume reassignment of roles is required. In such role based objects, Shuffler design pattern plays a significant place. The object should preserve its identity in this software system, where its identity is preserved but the roles assigned are shuffled. Hence objects of ShufflingObject class are truly mutable, since their children references are mutable as well [11].

2. **It supports variants in the shuffling of composite objects.** Shufflers make it easy to change the shuffling algorithm. The developer may just replace another ConcreteShuffler object. The shuffler may also consider few objects in the collection, which may be communicated by the client. Hence a part of the collection may be shuffled depending on the client's requirement.

3. **It supports refactoring.** The Shuffler pattern helps the designer in determining how to reorganize a design, which can reduce the effort taken to refactor the code at a later stage.

4. **The object is sliced or partitioned.** When two objects are swapped, only few of their data members which do not affect their identity are swapped. The remaining data members remain the same. This leads to violation of encapsulation, hence not suitable for applications where object is to be considered as a whole.

2.10 Implementation

Shuffler has many implementation variants and alternatives. Some important ones are given below. The trade-offs often depend on the control structures the programming language provides. Some languages (Java, for example) even support a variant of this pattern [9].

1. **The shuffling algorithm can be varied.** The AbstractShuffler makes it easy to change the shuffling algorithm, like Fisher-Yates algorithm, or Sattolo's algorithm, etc.

2. **The list to be shuffled can be dynamic.** The list can be selected by the client, at run-time. The client can provide the start and end of the list. Sometimes, if required a subset can be taken from the collection. The selection can be done based on the client's choice. The odd elements or even elements of the list may be considered for shuffling.

3. **Omitting the AbstractShuffler class.** There is no need to define an abstract Shuffler class when only one shuffle method is required. The abstract coupling that the Shuffler class provides lets the client have a privilege of deciding the shuffler method.

4. **The objects of the ShufflingObject can be linked as any suitable data structure.** Depending on the problem to which the pattern is applied, any suitable data structure like trees, hashes, dictionaries, tuples, etc. may be used.

2.11 Sample Code

The high level Java code for Jigsaw Shuffler redesigned with Shuffler pattern is listed in this section. The objective of this game is the end user will have to rearrange the jigsaw puzzle pieces in a meaningful manner. Each time the gamer starts a new instance of the puzzle, the shuffler module has to be invoked. The jigsaw puzzle program in Java was downloaded from the link [12]. The downloaded program was designed without patterns. The shuffler module was redesigned and implemented using the Shuffler pattern.

The PuzzleShuffler class listed below is an abstract class, which provides an interface for shuffling the puzzle pieces.

```
abstract class PuzzleShuffler{
    public ArrayList newInstance(ArrayList l){
        return l;
```

```
        }
        public void shuffle(ArrayList l){

        }
    };
```

The class JigsawShuffler which extends from PuzzleShuffler handles requests from the client for a shuffle. Also a request to add a new instance of a puzzle piece is also implemented in this class.

```java
class JigsawShuffler extends PuzzleShuffler{
    public ArrayList newInstance(ArrayList l){
            shuffle(l);
    return l;
    }
    public void shuffle(ArrayList l){
            Collections.shuffle(l);
    }
}
```

The client class PuzzleEx which is a JFrame is the UI from where the shuffle method is invoked. The code below here is partial, the complete implementation is not provided.

```java
public class PuzzleEx extends JFrame {
    private JPanel panel;
    private BufferedImage source;
    private ArrayList<MyButton> buttons;
    ArrayList<Point> solution = new ArrayList();
    private Image image;
    private MyButton lastButton;
    private int width, height;
    private final int DESIRED_WIDTH = 300;
    private BufferedImage resized;
    public PuzzleEx() {
            initUI();
    }
    private void initUI() {
            solution.add(new Point(0, 0));
```

```java
            solution.add(new Point(0, 1));
            solution.add(new Point(0, 2));
            solution.add(new Point(1, 0));
            solution.add(new Point(1, 1));
            solution.add(new Point(1, 2));
            solution.add(new Point(2, 0));
            solution.add(new Point(2, 1));
            solution.add(new Point(2, 2));
            solution.add(new Point(3, 0));
            solution.add(new Point(3, 1));
            solution.add(new Point(3, 2));
            buttons = new ArrayList();
            panel = new JPanel();
            panel.setBorder(BorderFactory.createLineBorder(Color.gray));
            panel.setLayout(new GridLayout(4, 3, 0, 0));
            try {
                    source = loadImage();
                    int h = getNewHeight(source.getWidth(), source.getHeight());
                    resized = resizeImage(source, DESIRED_WIDTH, h,
                    BufferedImage.TYPE_INT_ARGB);
            }
            catch (IOException ex) {
                    Logger.getLogger(PuzzleEx.class.getName()).log(
                    Level.SEVERE, null, ex);
            }
            width = resized.getWidth(null);
            height = resized.getHeight(null);
            add(panel, BorderLayout.CENTER);
            for (int i = 0; i < 4; i++) {
                for (int j = 0; j < 3; j++) {
                    image = createImage(new FilteredImageSource(resized.getSource(),
                    new CropImageFilter(j * width / 3, i * height / 4,
(width / 3), height / 4)));
                    MyButton button = new MyButton(image);
                    button.putClientProperty("position", new Point(i, j));
                    if (i == 3 && j == 2) {
                            lastButton = new MyButton();
```

```
                    lastButton.setBorderPainted(false);

                    lastButton.setContentAreaFilled(false);

                    lastButton.setLastButton();

                    lastButton.putClientProperty("position", new Point(i, j));

                    }
            else {

                    buttons.add(button);

                    }
        }
    }
    PuzzleShuffler pz=new JigsawShuffler();
    this.buttons = pz.newInstance(buttons);


    buttons.add(lastButton);
    for (int i = 0; i < 12; i++) {
        MyButton btn = buttons.get(i);
        panel.add(btn);
        btn.setBorder(BorderFactory.createLineBorder(Color.gray));
        btn.addActionListener(new ClickAction());
    }
    pack();
    setTitle("Puzzle");
    setResizable(false);
    setLocationRelativeTo(null);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
    }
```

2.12 Known Uses

Shufflers are common in object oriented systems. Several applications use shuffling to rearrange a collection of objects. They use different shuffling methods, but the objective is the same.

The code for the Jigsaw puzzle was analyzed. The library, named jqJigsawPuzzle in GitHub [4] also uses shuffle techniques. jqJigsawPuzzle is a JavaScript library that lets end users to create jigsaw puzzles in web pages. jqJigsawPuzzle.createPuzzleFromImage() is a method which creates a puzzle from a <img> element. The method takes another argument, *options*, which is optional. The *options* argument is an associative array with properties like, pieceSize

(which defines the size of the pieces, it can take the values 'normal', 'big' or 'small'), `borderWidth` (an integer which defines the width of the border around the puzzle) and `shuffle` (an associative array with the values `rightLimit`, `leftLimit`, `topLimit` and `bottomLimit`, which allow to extends the limits in which the pieces are moved when they are shuffled, since normally they are restricted to the puzzles frame). Here, each puzzle piece can be treated as an object of the JigsawPuzzlePiece class in Figure 1.

In the Sudoku game written in the Java programming language for Android, an ArrayList is filled with the numbers 1 to 9 and shuffled. Shuffling is important because otherwise the same puzzle will be returned. This code is available in the moulton package [5].

One of the most popular features of contemporary music software is the ability to randomize the order of the songs in a playlist—an action called "shuffling" the songs [6]. The user could be given the opportunity to shuffle the items, but then also come back to the ordered list. Shuffle mode may be turned on or off. The other feature is the user can shuffle within different types of music, like melody songs, party songs, devotional songs, etc. The typical algorithm is the Fisher-Yates shuffle. The result of the algorithm is a shuffled list, which is stored in memory. This list is then played from top to bottom, which enables the player to go backwards and forwards through the shuffled list. Usually, pressing 'stop' or closing the application will delete the shuffled list. Toggling the 'shuffle' button will wipe the shuffled list from memory and generate a new one.

Another example where shuffling is most commonly used is the Jumble Word Puzzle. In this puzzle, a set of scrambled letters are displayed. It also specifies a clue for discovering the underlying word(s). After solving the puzzle, a new set of letters and a clue will be displayed [7]. The source code is divided into three classes: HiddenWords, PuzzleBoard, WordClue. In the constructor of PuzzleBoard, a method scramble() is invoked, which has one argument, *word*. The word is scrambled in both forward and reverse directions. For example, if word contains "JavaWorld", scramWord must not contain "JavaWorld" or "dlroWavaJ" (must make it harder to guess).

2.13 Related Patterns

The related patterns to the shuffler pattern are:

- **Composite:** The composite objects can be shuffled by preserving the identity.

- **Iterator:** Iterators are applied to the composite Shuffling objects.

- **Command:** Shuffler can use Command to encapsulate a request as an object.

- **Strategy:** The Shuffling algorithm can be changed depending on the application need. The context (ShufflerManager) will decide on to which ConcreteStrategy (ConcreteShuffler) to create.

3. Conclusion

This paper contains a common design pattern that skilled object-oriented designers apply. Shuffling is immensely used in many applications worldwide. The Shuffler design pattern was not documented as an expertise of software practitioners. The paper describes the pattern in common vocabulary which the designers in future will use to communicate, document and explore design alternatives for the shuffling problem. This design pattern can

accommodate refactoring at design or code level with no trouble. As a future work, the influence of the Shuffler pattern on software quality attributes like reusability and maintenance can be studied.

Acknowledgements

References

[1]   Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Boston, 1994.

[2]   Ampatzoglou, A., Charalampidou, S., Stamelos, I., Research state of the art on GoF design patterns: A mapping study, *Journal of Systems and Software*, 86 (7), 1945-1964.

[3]   Patten, P. A., Howard, R. J., Stemmer, W. P. C., Applications of DNA Shuffling to pharmaceuticals and vaccines, *Current Opinion in Biotechnology*, 8 (6), 724-733.

[4]   GitHub Project: A JavaScript library that lets the end user create jigsaw puzzles in web pages. Retrieved November 10, 2015.
https://github.com/jfmdev/jqJigsawPuzzle.

[5]   MrBool Project: Java implementation for a mind game Sudoku. Retrieved November 11, 2015.
http://mrbool.com/how-to-implement-java-code-for-making-a-mind-game-sudoku/27013.

[6]   Reddit Post: How do shuffle algorithms work in music players such as VLC or iTunes. March 03, 2013, Retrieved November 12, 2015.
https://www.reddit.com/r/askscience/comments/19lf9t/how_do_shuffle_algorithms_work_in_music_players.

[7]   Friesen, J., Java World News: Hidden words puzzle game applet. March 6, 2007, Retrieved November 13, 2015.
http://www.javaworld.com/article/2077693/core-java/java-fun-and-games--puzzlemania.html

[8]   Alexander, C., Ishikawa, S., Silverstein, M., *A Pattern Language: Towns, Buildings, Construction*, Oxford, Britain, 2015.

[9]   Deitel, P., Deitel, H., *Java How to Program*, Prentice Hall, New Jersey, 2015.

[10]   Stackoverflow question: Combining Strategy and Composite pattern, October 15, 2013, Retrieved November 14, 2015.
            http://stackoverflow.com/questions/19383673/combining-strategy-and-composite-pattern.

[11]   Stackoverflow question: What are the consequences of immutable classes with references to mutable classes? June 23, 2011, Retrieved April 11, 2016.
            http://programmers.stackexchange.com/questions/86254/what-are-the-consequences-of-immutable-classes-with-references-to-mutable-classe.

[12]   ZetCode Tutorial: The Puzzle game, February 19, 2015, Retrieved April 22, 2016.
            http://zetcode.com/tutorials/javagamestutorial/puzzle/