Static Code Analysis

November 19, 2015

Kurt E. Clothier
Lovedeep Gondara
Kyle Kampfen
[Group 5]

CSC 578 B — Software Engineering
University of Illinois Springfield
Instructor: West, Roger

These are all issues that were found in our source code. Some issues were found using the tool PMD integrated into Eclipse and are noted as such. Other issues were found using either static or dynamic analysis. In all cases, the refactored code was tested for proper operation using test benches. The code is all written in Java, with select fragments shown where appropriate.

Issue 1: PMD Rule: ClassCastExceptionWithToArray

There are various instances in the program where it is desirable to return an array from a collection of Playing Card objects (or any object, for that matter), but a compilation error prevents a simple assignment using *Collection.toArray()* without casting the value as an array of the desired object. This compiles fine, but will throw a *ClassCastException* at runtime. The simple solution is to pass an array of the returned object type to the *toArray()* method.

Original method:

```java
public PlayingCard[] getAll() {
	return (PlayingCard[])this.cards.toArray();
}
```

Solution:

```java
public PlayingCard[] getAll() {
	return this.cards.toArray(new PlayingCard[cards.size()]);
}
```

Issue 2: PMD Rule: UseVarargs

In addition to returning an array of Playing Card objects, sometimes, it is necessary to pass an array of Playing Cards in order to move them from one pile to another. Rather than using separate methods to add a single card or an array of cards, the Java varargs notation can be used in either case. This reduces the number of methods necessary for the same operation, making the class easier to maintain.

Original methods:

```java
public void add(final PlayingCard card) {
	cards.add(card);
}

public void add(final PlayingCard[] cards) {
	for (final PlayingCard card : cards){
		this.cards.add(card);
	}
}
```

Solution:

```java
public void add(final PlayingCard...cards) {
	for (final PlayingCard card : cards){
		this.cards.add(card);
	}
}
```

Issue 3: Plugin as a “god class”

One of the central pieces of the card game engine is the *Plugin* – a class representing an individual engine plugin. Currently, plugins can be of type DECK, BOARD, or RULES, but this list can be easily expanded. The real issue is that because the *Plugin* is such an integral component, it has quickly ballooned into a rather large and unmaintainable entity, commonly known as a “god class.” The current implementation of the Plugin class contained over 600 lines of code, which is not all that much, but also contained were a collection of 8 public static attributes, 4 public constructors, 20 public methods (some of which are factory-like), 12 public static methods (some of which are factory-like), and a private inner class used by some of the static methods. Of course, this list only grows as new functionality is found necessary and added.

```java
public class Plugin {

	public static enum Type { DECK, BOARD, RULES }

	 // Lots of methods…
}
```

There are a few possibly solutions to this issue. The first is to create a separate final *PluginUtil* class with nothing but the static *Plugin* related utilities. The main class could also be broken into smaller classes with more specific purposes to follow the single responsibility principle. This is a plausible solution, but can require the creation of numerous copies of the same object, each with its own ability but high coupling between the methods of these classes. In the end, this issue was resolved by splitting the main class into a few pieces, with methods that only affected those pieces available in the new classes. Splitting into these pieces also aided in type and null checking which could be done once for each issue.

Another issue with this class is its location. This class is contained in the same package as the other game engine utilities, although it doesn’t have anything to with any of them. Some components will use a *Plugin* during creation, but they should seek this class out; it has no need to live with them in the same package. The fix for this is to simply move *Plugin* related classes to a new package and update any calls to it.

Solution (fragment):

```java
public class Plugin {
	// Reduced to 400 LoC
	// Reduced to 17 methods
	// Removed all static methods
}

public final class PluginFilename {
	// Handles all file name (String) getters and type checking
}

public final class PluginFile extends File {
	// Handles all file getters and type checking
}

public final class PluginList {
	// Single Purpose class for returning lists of file names
}

public enum PluginPattern {
	 // Holds all String constants for pattern matching
}
```

Issue 4: Increasing Code Reuse and Type Safety

Aside from issues in the *Plugin* Class itself, there are numerous problems with how it is being used. For example, when looking for a particular Keyword in a file, something like the following was used:

```
String parameters = plugin.getParams("some-keyword");
if (parameters != null && parameters.equals("some-value")) {/*...*/}
else if (parameters != null && parameters.equals("some-other-value")) {/*...*/}
```

This method will return any parameters (as a String) found after the given keyword, and the parameter is then checked against known possible strings. The first obvious improvement is to move all of these String literals to *private static final* variables, but that doesn't really solve the bigger issues of constantly checking against Strings and null. It also makes it very difficult to add functionality to the system with new keywords or valid parameters. The next step was to create enumerations with valid keywords and parameters such as:

```
public static enum Orientation { LANDSCAPE, PORTRAIT }
```

In this example, *Orientation* is a keyword, and the enum members are valid parameters. This introduces type safety and a known reference set of valid keywords. Checking if a plugin has valid parameters is now done as:

```
Orientation orientation = Orientation.valueOf(plugin.getParams(SOME_KEYWORD).toUpperCase());
```

This new version can only return a valid Orientation member which can be used in a Switch statements rather than if-else blocks. The downside to this is that if the parameters returned after *SOME_KEYWORD* are not actually a valid member, an exception is thrown. This can be captured in a try-catch block and transformed into a custom *PluginException*, but that is a lot of code to write for every single keyword interaction. Some more refactoring will completely solve this issue. Now, every possible keyword used in a *Plugin* file is kept in an enum called *PluginKeyword*. Methods involving these keywords can now be accessed from this enum, and a special method was created to ensure keyword-parameter type safety:

```
public enum PluginKeyword {
	ORIENTATION;  // One of many keywords…

	public String checkParams(final Plugin plugin) throws PluginException {
		return plugin.checkParams(this).toUpperCase(Plugin.LOCALE);
	}

	public <T extends Enum<T>> T checkBoundedParams(final Plugin plugin,
								final Class<T> enumType) throws PluginException {
		T enumMember = null;
		try {
			enumMember = Enum.valueOf(enumType, this.checkParams(plugin));
		} catch (IllegalArgumentException e) {
			throw PluginException.create(PluginException.Type.INVALID_PARAMETER, e);
		}
		return enumMember;
	}
}
```

Finally, keyword parameters are found as such, without the need for null checks or exception handling:

```
Orientation orientation = PluginKeyword.ORIENTATION.checkBoundedParams(plugin, Orientation.class);
```

Notice, the use of a *checkParams()* rather than *getParams()* method means an exception is thrown if the keyword is not found such that this method adheres to the "fail fast" philosophy.

Issue 5: PMD Rules: SimplifyBooleanReturns, UseLocaleWithCaseConventions

When a method is designed to return a Boolean, using numerous if-then statements is redundant, since they use Boolean expressions to execute. The following method is the overridden *accept()* method from a class implementing the *FilenameFilter* interface. It returns true if the passed filename is not filtered out. Simplifying this type of Boolean method also solves the *controversial* issue of multiple return statements in a single method, and the reasoning to not use braces for the single line if-then statements. Another small issue in this block of code is the use of the *toLowerCase()* method which will work somewhat differently for different locales; hence, a locale should be specified. In addition, an increase in performance can be found by declaring the parameters to be final. In these methods, the *prefix* parameter is a private field set in the constructor.

Original method:

```java
@Override public boolean accept(File dir, String name) {
        String nameLC = name.toLowerCase();
        if (!nameLC.startsWith(prefix))
                return false;
        if (!nameLC.endsWith(Plugin.EXTENSION))
                return false;
        if (nameLC.contains("template"))
                return false;
        if (nameLC.contains("test"))
                return false;
        return true;
}
```

Solution:

```java
@Override public boolean accept(final File dir, final String name) {
        final String nameLC = name.toLowerCase(Plugin.LOCALE);
        return          nameLC.startsWith(prefix) &&
                        nameLC.endsWith(Plugin.EXTENSION) &&
                        !(      nameLC.contains("template") ||
                                nameLC.contains("test"));
}
```

Issue 6: Efficiently copying arrays

There are numerous components that use arrays to internally store objects which have no use for any of the benefits of an ADT collection. One such example is the *Deck* object, which only needs to access a contained *PlayingCard* at a known index. The issue comes in constructing the deck. It doesn't make sense to create a *Collection* of cards just for them to be turned into an array inside of the *Deck*, so why not just pass an array of cards to the deck? We also know that we want the deck to reference the actual passed cards, not copies of them. In this case, a *PlayingCard* is immutable, but in other cases, we actually want changes made to the passed objects to be reflected outside of the method or class. In any case, creating new objects as exact copies needlessly wastes memory.

Original constructor (with irrelevant parameters and operations removed):

```java
public Deck(final PlayingCard...cards) {
        this.size = cards.length;
        // Copy cards into this deck
        this.cards = new PlayingCard[this.size];
        for (int i = 0; i < this.size; i++) {
                this.cards[i] = cards[i];
        }
        this.dealNdx = 0;  // Array index
}
```

This original operation works perfectly fine. A new array is created with the same size as cards parameter, and the contents of the cards parameter are copied into the new array. Note, this is a shallow copy – the actual *PlayingCards* contained in the cards parameter will be referenced in the new cards array. There are no new *PlayingCards* being constructed. This is what we want; however, there are other, native ways of doing this.

Option 1: Clone the array

```java
public Deck(final PlayingCard...cards) {
      this.size = cards.length;

       // Copy cards into this deck
       this.cards = cards.clone();
      this.dealNdx = 0;  // Array index
}
```

This option works, but it uses the dreaded *clone()* method. In this case, it is OK. The clone() method does a shallow copy, so it is merely copying the references of each *PlayingCard* in the cards parameter to the new cards array. In other words, changes made to the contained objects of one array will be reflected in the other, but changes to the array itself (such as *this.cards[2] = new PlayingCard()*) will not. Again, in this case it doesn't matter because both the *cards* parameter and *this.cards* are final, and a *PlayingCard* is immutable.

Option 2: Using System.arraycopy()

```java
public Deck(final PlayingCard...cards) {
      this.size = cards.length;

       // Copy cards into this deck
       this.cards = new PlayingCard[this.size];
       System.arraycopy(cards, 0, this.cards, 0, this.cards.length);
      this.dealNdx = 0;  // Array index
}
```

Similar to the the clone() method, this build in method can be used for creating shallow copies of an array. The primary advantage is that this method is Java Native, so it is generally thought to perform better using the JNI. Even still, there are disadvantages of using the *System.arraycopy()* method. First of all, there are a lot of parameters to keep track of between the source and destination arrays, the start indices, and the number of objects to be copied. Lastly, this method copies to an existing array hence the need to first instantiate the new array. This method is fast and powerful, but to simply copy the entire array, there may be an easier method.

Option 3: Using Arrays.copyOf()

```java
public Deck(final PlayingCard...cards) {
      this.size = cards.length;

       // Copy cards into this deck
       this.cards = Arrays.copyOf(cards, cards.length);
      this.dealNdx = 0;  // Array index
}
```

This final option combines the best of the previous methods. It is built off of the previous *System.arraycopy()* native method, but it reduces the number of parameters and returns a new array similar to *clone()*, alleviating the need to instantiate one manually. In addition, it is easy to read and the intent is obvious. It is for these reasons that this method will be used to create shallow-copied arrays as opposed to a manual loop or other in-built methods.

Issue 7: Memory Usage – Lots of duplicate objects

In the original design, our card game engine created unique *PlayingCard* objects. These objects were truly unique, as each was given a unique ID used for equality comparisons. The reasoning for this was to be able to test if a specific card was contained in one player's hand (or other pile) or not; however, it was later decided that this ability was not as important as checking if a player held a card with specific attributes. In addition, creating unique objects for every Playing Card would be very inefficient for games such as Blackjack which typically contain numerous copies of the same card. To solve this issue, a factory pattern was used in order to implement the flyweight pattern - create playing cards as they are needed as well as providing reference to previously created playing cards with the desired attributes when appropriate.

Combination solution is shown for issues 7-9.

Issue 8: Unnecessary Coupling Between Engine Components

In addition to the previous issue, the *PlayingCard* class was needlessly coupled to a card ranking system. Each card held an attribute representing a numeric value of that card. This is value is not something the playing card needs to keep track of, and different games might use a different type of ranking system anyway. The solution to this was to remove the *value* attribute of each card and create an additional *Ranking* class which will provide a numeric value for a given card. This Ranking object can be created to rank cards in various ways, depending upon the game being played; however, neither the *PlayingCard* nor the *Ranking* needs to be aware that any sort of game actually exists, and *PlayingCard* doesn't need to know there is any sort of ranking.

Combination solution is shown for issues 7-9.

Issue 9: Attribute Readability and Null Values

The last issue involving the *PlayingCard* class is the use of Strings to represent some of the attributes. One issue is the readability of this, and the potential for mixing the face and group attributes. Wrapping these Strings into Attribute classes can solve these issues as well as separating the comparison logic among the individual attributes rather than being fully contained in the *PlayingCard* class. Another issue involving these attributes is the use of Null values. One null value is OK, but it is undesirable for a Card to be created with all null attributes. If this happens, an exception is thrown. This will greatly reduce the number of null checks which must be performed on *PlayingCard* attributes. Each attribute also implements a similar flyweight factory pattern similar to the *PlayingCard* class.

Combination solution is shown for issues 7-9.

Original PlayingCard Class (fragment):

```java
public final class PlayingCard implements Comparable<PlayingCard>, Serializable, Cloneable {

	private static final AtomicInteger UNIQUE_ID = new AtomicInteger(0);
	private final int id;
	// other attibutes

	protected PlayingCard(final String face, final int value, final String group) {
		this.face = face;
		this.value = value;
		this.group = group;
		this.id = UNIQUE_ID.getAndIncrement();
	}
}
```

Solution (fragment):

```
public final class PlayingCard implements Comparable<PlayingCard>, Serializable, Cloneable {

	private final Face face;
	private final Group group;

	PlayingCard(final Face face, final Group group) throws IllegalArgumentException {
		if (face == null && group == null ) {
			throw new IllegalArgumentException("Both parameters cannot be null!");
		}
		this.face = face;
		this.group = group;
	}
}

// Playing Card Singleton Flyweight Factory
public enum PlayingCardFactory {

	INSTANCE;

	private final List<PlayingCard> cards;

	private PlayingCardFactory() {
		cards = new ArrayList<PlayingCard>();
	}
	public static PlayingCardFactory getInstance()
	{
		return INSTANCE;
	}

	public synchronized PlayingCard createPlayingCard(final Face face, final Group group) {
		PlayingCard card = null;
		for (final PlayingCard p : cards) {
			if	(p.has(face, group)) {
				card = p;
				break;
			}
		}
		if (card == null) {
				card = new PlayingCard(face, group);
				cards.add(card);
		}
		return card;
	}
}

// Playing Card Ranking
public final class PlayingCardRanking implements Serializable, Comparator<PlayingCard> {
	 // TODO
}
```

Issue 10: PMD Rules: AppendCharacterWithChar, ConsecutiveAppendsShouldReuse

When using StringBuilders, there are a lot of tricks that be done. In this case, a single character should not be concatenated as a string. We can also conform to the *fluent interface* style, promoted by Martin Fowler, by reusing the target object for consecutive appends as opposed to individual assignments. The following method is used to create a valid *Plugin* filename given a *type* of plugin file and the name sans type or extension. Similar to previous examples, variables are made *final* where applicable.

Original method:

```
public static String createFilename(Plugin.Type type, String rawName) {
      StringBuilder str = new StringBuilder();
      str.append(type.toString().toLowerCase());
      str.append(".");    // <-- This char is treated as a String
      str.append(rawName);
      str.append(Plugin.EXTENSION);
      return str.toString();
}
```

Solution:

```
public static String createFilename(final Plugin.Type type, final String rawName) {
      final StringBuilder str = new StringBuilder();
      str.append(type.toString().toLowerCase(Plugin.LOCALE))
         .append('.').append(rawName).append(Plugin.EXTENSION);
      return str.toString();
}
```

Issue 11: PMD Rules: AvoidCatchingNPE, AvoidCatchingGenericException, ExceptionAsFlowControl

In Java, exceptions are incredibly important. In addition to being an invaluable tool in trouble shooting, they can also be thrown on purpose in order to "fail fast" or otherwise explain why something isn't working. In this program, the various types of exceptions recurring as a result of an ill formatted plugin file should be consolidated in order to alert the user of the problem stemming from an external text file as opposed to internal code. However, it is fairly easy to abuse this ability, and one such example is catching a NullPointerException. These types of exceptions typically mean something is wrong, and ignoring them will likely result in another unhandled exception somewhere else. This particular type of exception handling can also be seen as using exceptions to control the flow of code, as there are alternatives which do not involve exception handling.

In the following code fragment, the *rank* of a card is being acquired from a *Map<String, Integer>* of the card rankings. The rankings key is a String – the face name of the card (such as "king"), and the integer returned is the rank of the card. The String keys are contained an array called *faces*, and accessed with *faceNdx*. A *NullPointerException* here indicates that there is no mapping for specified card face name; hence, there is an issue with the plugin file specifying the ranking of the playing cards, and a special *PluginException* should be thrown. Rather than catch the *NPE*, we should try to prevent it.

Original method (fragment):

```
try {
      rank = ranking.get(faces[faceNdx]);
} catch (NullPointerException e) {
      throw PluginException.create(PluginException.Type.MISMATCH, e, deck.getName(), "Ranking");
}
```

Solution:

```
if (ranking.containsKey(faces[faceNdx])) {
      rank = ranking.get(faces[faceNdx]);
}
else {
      throw PluginException.create(PluginException.Type.MISMATCH, deck.getName(), "Ranking");
}
```