OTP-GROUP 6 Statistical code review

Introduction

This document summarizes and discusses results from 3 code analysis tools: Checkstyle, PMD, and SpotBugs, on a Java-based dating application. Key areas to cover include code violations and errors, optimizations, and readability and formatting issues. Results are presented under a dedicated section for each tool with screenshots to support them. The results are then summarized with the goal of highlighting high-priority areas and issues, and potential ways to resolve them.

Key Metrics

Below are four important metrics to evaluate the complexity and health of a codebase. Addressing them is crucial to make code easier to maintain, improve optimization and readability, and follow best practices.

Code Complexity (Cyclomatic Complexity)

Cyclomatic complexity measures the number of independent paths through a program's source code, which indicates how difficult the code is to test and maintain.

- **Measurement**: The number of decision points (if statements, loops, etc.) plus one
- **Importance**: Higher complexity = higher risk of bugs and maintenance difficulties
- **Typical threshold**: A complexity above 10-15 is often considered concerning
- **Example**: A function with 3 if statements would have a cyclomatic complexity of 4 (3+1)

Lines of Code per Method

This metric counts the physical lines of code within individual methods or functions.

- **Measurement**: The raw size of methods in your codebase
- **Importance**: Excessively long methods are harder to understand and maintain
- **Typical threshold**: Methods exceeding 20-30 lines often benefit from refactoring
- **Best practice**: Methods should do one thing well, not many things poorly

Duplicate Code

This identifies repeated code patterns across a codebase.

- **Measurement**: Code segments that appear multiple times with little or no variation
- **Importance**: Violates the "Don't Repeat Yourself" (DRY) principle
- **Problems caused**:
  - Bug fixes need to be applied in multiple places
  - Increases codebase size unnecessarily
  - Makes maintenance more difficult
- **Detection**: Usually measured as a percentage of duplicated lines

Potentially Unreachable Code Blocks

This identifies code that can never be executed during runtime.

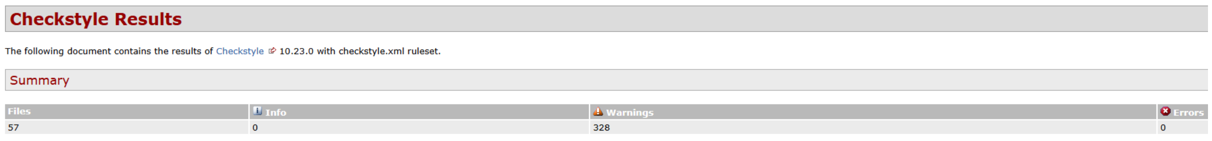
- **Measurement**: Code sections that have no execution path that leads to them
- **Importance**:
  - Indicates logical errors in program flow
  - Creates confusion for developers
  - Bloats the codebase with useless instructions
- **Common causes**:
  - Conditions that always evaluate to false
  - Code after unconditional return statements
  - Unreachable case statements

Checkstyle

This section covers the initial results of Checkstyle, a tool that checks java code to enforce a coding standard. This coding standard is highly configurable to comply with a project's specific requirements. The tool's main goals are to improve readability, make formatting consistent, and point out class and method design problems, which can address code violations and errors and improve optimization. This project uses a slightly modified version of Google's Java Style Checkstyle configuration, with the main changes being made to not include missing JavaDoc comments, and to account for cyclomatic complexity and lines of code per method.

Results

Before any refactoring or code cleanup, the CheckStyle tool reported a total of 328 warnings and 0 errors. This means the code doesn't have any errors, but contains many parts with inconsistent formatting or best practice violations, at least according to Google's configuration. Therefore, the readability of the code could be improved. The results summary can be seen in image 1.



Checkstyle Results

The following document contains the results of Checkstyle 10.23.0 with checkstyle.xml ruleset.

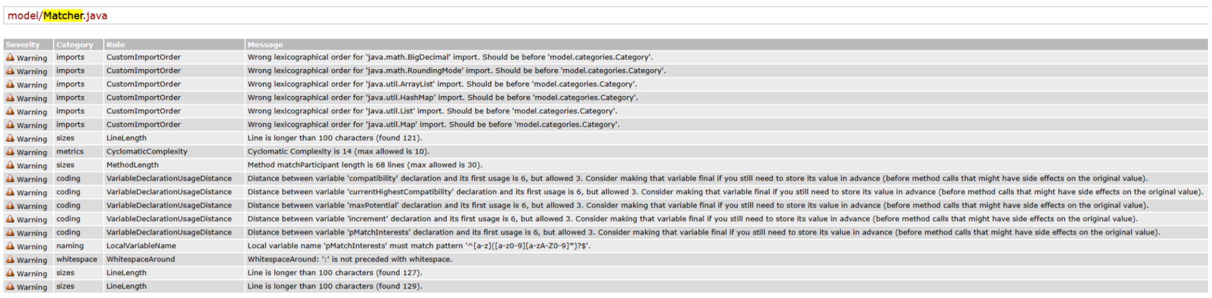
Summary

Files	Info	Warnings	Errors
57	0	328	0

Image 1. Checkstyle results summary.

Findings

Warning categories ranged between naming, imports, coding, sizes, whitespace, and blocks. The import category seems by far the most common, with most issues being related to import statements being in the wrong lexicographical order. Some other common issues were lines being longer than 100 characters and variables not being declared in their own statements. Notably, 2 warnings were related to a cyclomatic complexity of over 10 and 7 to methods containing over 30 lines of code. Image 2 showcases a class with multiple different warnings.



model/Matcher.java

Severity	Category	Rule	Message
Warning	imports	CustomImportOrder	Wrong lexicographical order for 'java.math.BigDecimal' import. Should be before 'model.categories.Category'.
Warning	imports	CustomImportOrder	Wrong lexicographical order for 'java.math.RoundingMode' import. Should be before 'model.categories.Category'.
Warning	imports	CustomImportOrder	Wrong lexicographical order for 'java.util.ArrayList' import. Should be before 'model.categories.Category'.
Warning	imports	CustomImportOrder	Wrong lexicographical order for 'java.util.HashMap' import. Should be before 'model.categories.Category'.
Warning	imports	CustomImportOrder	Wrong lexicographical order for 'java.util.List' import. Should be before 'model.categories.Category'.
Warning	imports	CustomImportOrder	Wrong lexicographical order for 'java.util.Map' import. Should be before 'model.categories.Category'.
Warning	sizes	LineLength	Line is longer than 100 characters (found 121).
Warning	metrics	CyclomaticComplexity	Cyclomatic Complexity is 14 (max allowed is 10).
Warning	sizes	MethodLength	Method matchParticipant length is 68 lines (max allowed is 30).
Warning	coding	VariableDeclarationUsageDistance	Distance between variable 'compatibility' declaration and its first usage is 6, but allowed 3. Consider making that variable final if you still need to store its value in advance (before method calls that might have side effects on the original value).
Warning	coding	VariableDeclarationUsageDistance	Distance between variable 'currentHighestCompatibility' declaration and its first usage is 6, but allowed 3. Consider making that variable final if you still need to store its value in advance (before method calls that might have side effects on the original value).
Warning	coding	VariableDeclarationUsageDistance	Distance between variable 'maxPotential' declaration and its first usage is 6, but allowed 3. Consider making that variable final if you still need to store its value in advance (before method calls that might have side effects on the original value).
Warning	coding	VariableDeclarationUsageDistance	Distance between variable 'increment' declaration and its first usage is 6, but allowed 3. Consider making that variable final if you still need to store its value in advance (before method calls that might have side effects on the original value).
Warning	coding	VariableDeclarationUsageDistance	Distance between variable 'pMatchInterests' declaration and its first usage is 6, but allowed 3. Consider making that variable final if you still need to store its value in advance (before method calls that might have side effects on the original value).
Warning	naming	LocalVariableName	Local variable name 'pMatchInterests' must match pattern '^[a-z]([a-z0-9][a-zA-Z0-9]*)?$'.
Warning	whitespace	WhitespaceAround	WhitespaceAround: ':' is not preceded with whitespace.
Warning	sizes	LineLength	Line is longer than 100 characters (found 127).
Warning	sizes	LineLength	Line is longer than 100 characters (found 129).

Image 2. Matcher class warning summary and descriptions.

Discussion

In summary, the readability and consistency of the source code could be improved drastically through refactoring and cleanup. Furthermore, 2 methods had a cyclomatic complexity of over 10, and 7 methods were too long according to our standards. Therefore, the code's complexity could be slightly improved, while several methods could benefit from refactoring and breaking down into smaller pieces.

To improve the readability of the code, automated tools like OpenRewrite can be used to reformat it by integrating them with the project's maven build configuration. The project's formatting can be further improved by providing IntelliJ Idea, our IDE, the checkstyle configuration. It can then reformat the code with a single button press. However, some issues will still be present, especially those related to complexity and method length, requiring manual inspection and fixes.

PMD

PMD is a multi-language static code analyzer, meaning it does not execute the code. Instead, it analyzes the source code to find common programming flaws such as unused variables, empty *catch* blocks, and unnecessary object creation by checking against predefined or custom "Rulesets". PMD automates the process of code review by detecting bugs, syntax errors, and potential security issues.

Findings

Issue Type	**Count**	**Description**
SystemPrintln	~70+	Use of *System.out.println* or *System.err.println* - should be replaced with more robust logging
OneDeclarationPerLine	~15	Declaring multiple variables in one line
UnusedPrivateMethod	~10	Private methods not being used, "dead" code
UnusedFormalParameter	~10	Method parameters are declared but never used
LiteralsFirstInComparisons	~8	Strings not positioned first in *.equals()* may result in *NullPointerException*
LooseCoupling	3	Using *HashMap* directly instead of *Map*
AvoidReassigningParameters	2	Method parameters are reassigned
NonExhaustiveSwitch	1	Missing *default* in a *switch*, could lead to logical bugs

Table 1. List of PMD findings.

Discussion

The static code analysis with PMD revealed 122 maintainability and style issues. The most common ones were caused by the overuse of *System.out.println*, unused methods and parameters, as well as readability issues like multi-variable declarations in one line. Addressing most of these will significantly improve the maintainability and professionalism of the code.

Our development team used a fair amount of console printing for debugging purposes, and these have not been cleaned up after the issues were resolved. Additionally, some catch blocks only log error messages to the console, where a more sophisticated error-handling approach would be better.

Some of the multi-variable declarations could be cleaned up to be in separate lines. However, in JavaFX views, grouping similar elements in one declaration can be more efficient in certain cases. The following example declares multiple *Label* elements used in the FXML file. Refactoring these into separate lines would not improve clarity and may even reduce readability in this context.

```
@FXML
private Label percentageLabel, matchParticipantsLabel, interestsLabel;
```

PMD flags the following line due to tight coupling to a specific implementation class:

```
HashMap<User, Double> topMatches = matcher.getTopMatches();
```

Refactoring it to use the interface instead improves flexibility and abstraction:

```
Map<User, Double> topMatches = matcher.getTopMatches();
```

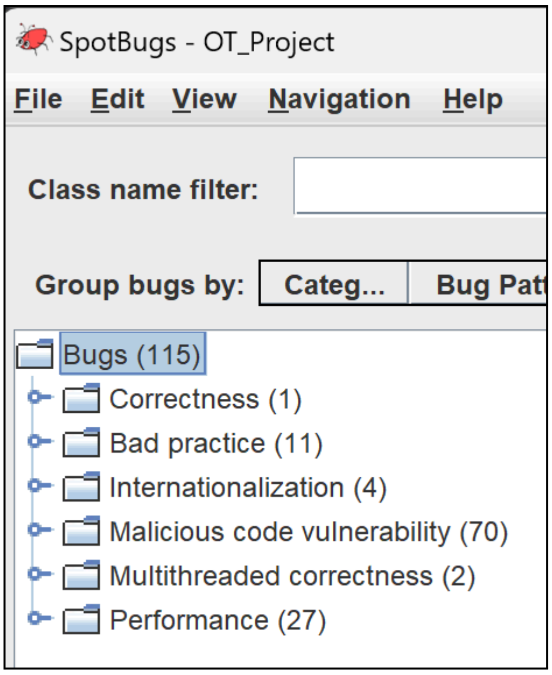
This simple change allows for easier maintenance and future enhancements, such as swapping the underlying implementation without impacting the rest of the codebase. These types of issues are particularly valuable to address, as they are easy to fix yet improve code quality and design. Many unused private methods and parameters suggest some blocks are not used, and these are to be refactored.

SpotBugs

SpotBugs is a tool that analyzes Java code to find bugs using static analysis. It examines compiled code for bugs without actually running the code. SpotBugs looks for bug patterns, which are pieces of code that are likely to contain errors. These bug patterns could lead to errors, security vulnerabilities, or poor performance. SpotBugs is a fork of FindBugs, which is an abandoned project. SpotBugs is a continuation of FindBugs and it is developed with the support of its community. It is free software and can be used, modified, and distributed under its open-source license.

Results

After running “mvn clean compile spotbugs:spotbugs” and “mvn spotbugs:gui” on the project, the GUI report yielded a total of 115 bugs. Image 3. shows the GUI with the number of bugs and their subcategories.


SpotBugs - OT_Project

File Edit View Navigation Help

Class name filter:

Group bugs by: Categ... Bug Patt

- Bugs (115)
  - Correctness (1)
  - Bad practice (11)
  - Internationalization (4)
  - Malicious code vulnerability (70)
  - Multithreaded correctness (2)
  - Performance (27)

Image 3. SpotBugs GUI after running spotbugs on compiled code.

Bug Descriptions

The subcategories for the bugs found in our project are **correctness**, **bad practice**, **malicious code vulnerability**, **performance**, and **dodgy code**. These categories are defined as follows on the SpotBugs bug description documentation.

Correctness: Probable bug - an apparent coding mistake resulting in code that was probably not what the developer intended. We strive for a low false positive rate.

Bad practice: Violations of recommended and essential coding practice. Examples include hash code and equals problems, cloneable idiom, dropped exceptions, Serializable problems, and misuse of finalize. We strive to make this analysis accurate, although some groups may not care about some of the bad practices.

Internationalization: Code flaws having to do with internationalization and locale.

Malicious code vulnerability: Code that is vulnerable to attacks from untrusted code.

Multithreaded correctness: Code flaws having to do with threads, locks, and volatiles.

Performance: Code that is not necessarily incorrect but may be inefficient.

These categories give an overview of the type of bugs found in the initial state of our project. Most prominent subcategory is malicious code vulnerability with 70 bugs in total. The other categories such as correctness contained 1, bad practice 11, internationalization 4, multithreaded correctness 2, and performance 27, which is 115 bugs in total.

Findings

Correctness

The single bug in **correctness** had to do with handleMatchClick, where a potential null pointer dereference could happen because of using match after a null check without an early return as seen in image 4.

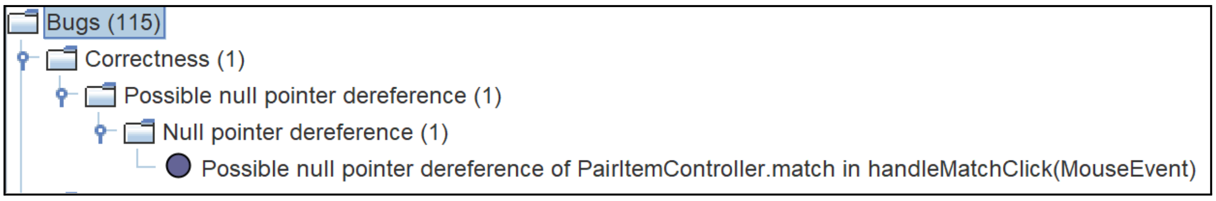
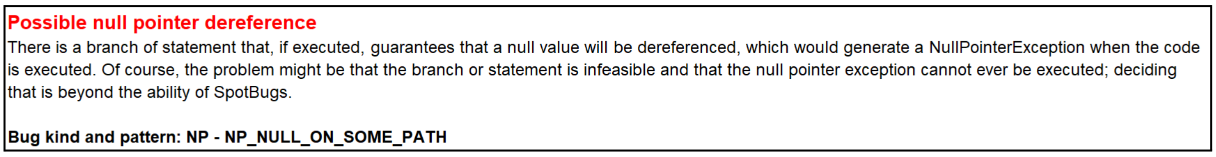

Bugs (115)
Correctness (1)
Possible null pointer dereference (1)
Null pointer dereference (1)
Possible null pointer dereference of PairItemController.match in handleMatchClick(MouseEvent)

Image 4. SpotBugs GUI subcategory correctness containing one bug.

A better description of the issue is received in the GUI after clicking an instance of the bug as seen in image 5.


Possible null pointer dereference
There is a branch of statement that, if executed, guarantees that a null value will be dereferenced, which would generate a NullPointerException when the code is executed. Of course, the problem might be that the branch or statement is infeasible and that the null pointer exception cannot ever be executed; deciding that is beyond the ability of SpotBugs.

Bug kind and pattern: NP - NP_NULL_ON_SOME_PATH

Image 5. description of the bug dealing with null pointer dereference.

Bad Practice

The **bad practice** category had 11 bugs as seen in image 6. 8 of these 11 bugs are described as methods that intentionally throw RuntimeException. These bugs were found in the following classes GuestDao, MatchDao, UserDao. 3 bugs had to do with getResource() being used in a subclass, which after further extension could lead to unexpected behavior the issue is described in image 8. This was violated in the following parts of the code AdminHomeController.createMatchItemNode(Match), AdminUsersController.createUserItemNode(User), and view.GUI.start(Stage).

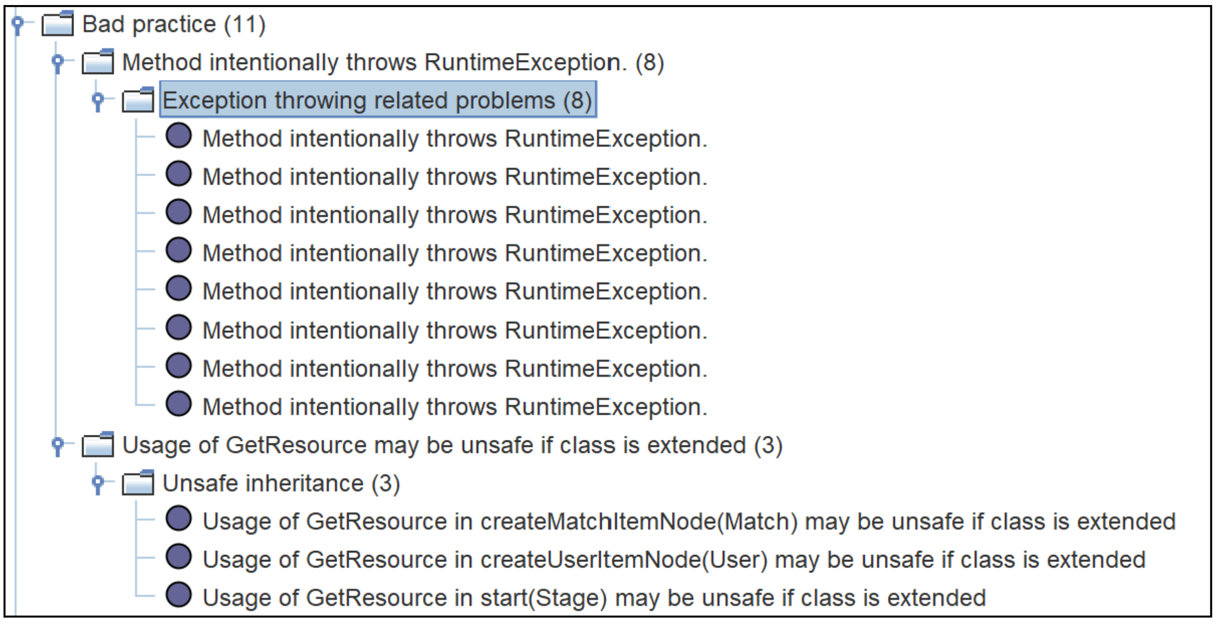

Bad practice (11)
Method intentionally throws RuntimeException. (8)
Exception throwing related problems (8)
Method intentionally throws RuntimeException.
Method intentionally throws RuntimeException.
Method intentionally throws RuntimeException.
Method intentionally throws RuntimeException.
Method intentionally throws RuntimeException.
Method intentionally throws RuntimeException.
Method intentionally throws RuntimeException.
Method intentionally throws RuntimeException.
Usage of GetResource may be unsafe if class is extended (3)
Unsafe inheritance (3)
Usage of GetResource in createMatchItemNode(Match) may be unsafe if class is extended
Usage of GetResource in createUserItemNode(User) may be unsafe if class is extended
Usage of GetResource in start(Stage) may be unsafe if class is extended

Image 6. SpotBugs GUI subcategory bad practice containing 11 bugs.

The bug with a method intentionally throwing a RuntimeException was described as follows in the GUI image 7. after selecting the bugs.

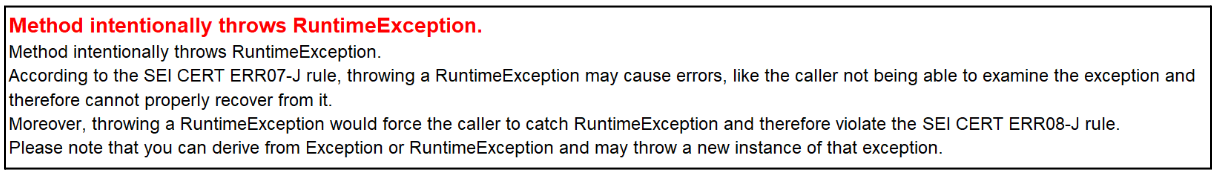
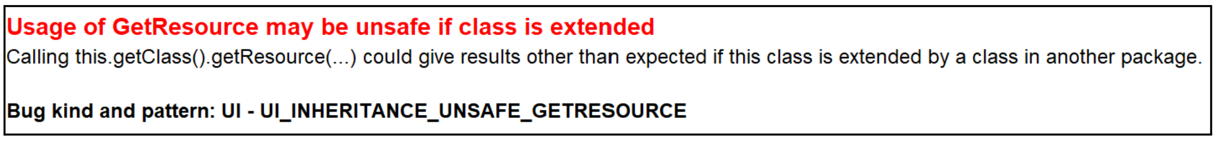

Method intentionally throws RuntimeException.
Method intentionally throws RuntimeException.
According to the SEI CERT ERR07-J rule, throwing a RuntimeException may cause errors, like the caller not being able to examine the exception and therefore cannot properly recover from it.
Moreover, throwing a RuntimeException would force the caller to catch RuntimeException and therefore violate the SEI CERT ERR08-J rule.
Please note that you can derive from Exception or RuntimeException and may throw a new instance of that exception.

Image 7. description of the issue dealing with methods intentionally throwing RuntimeException.

The issue dealing with the unsafe inheritance if class is extended was described as follows in the GUI seen in image 8.


Usage of GetResource may be unsafe if class is extended
Calling this.getClass().getResource(...) could give results other than expected if this class is extended by a class in another package.

Bug kind and pattern: UI - UI_INHERITANCE_UNSAFE_GETRESOURCE

Image 8. description of the issue with unsafe inheritance if class is further extended.

Internationalization

In the **internationalization** category there were 4 bugs as seen in image 9. Further subcategories give more description of the type of bugs at hand. Final category being about dubious methods used. The issues lie with several instances of strings being converted to uppercase or lowercase. This can lead to incorrect behavior with non-English languages due to locale specific casing rules. These bugs are found in the following parts of the code AfterMatchController.setResults(), InterestSelectionController.createOptionPane(Category), ProfileController.initialize(), and MatchUtils.lambda$formatInterests$0(String).

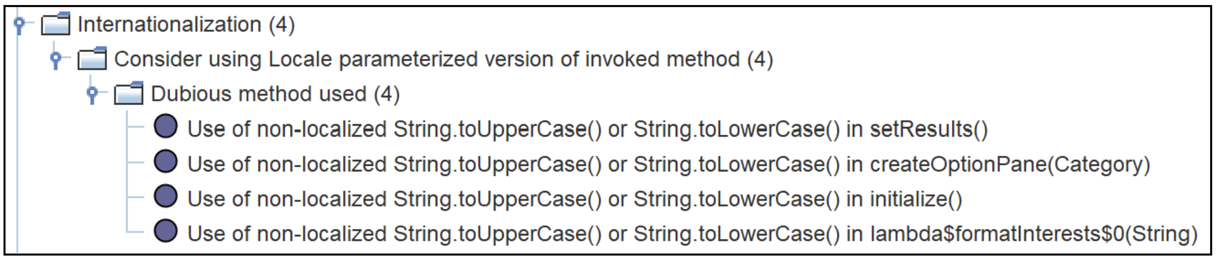

```
Internationalization (4)
  Consider using Locale parameterized version of invoked method (4)
    Dubious method used (4)
      ● Use of non-localized String.toUpperCase() or String.toLowerCase() in setResults()
      ● Use of non-localized String.toUpperCase() or String.toLowerCase() in createOptionPane(Category)
      ● Use of non-localized String.toUpperCase() or String.toLowerCase() in initialize()
      ● Use of non-localized String.toUpperCase() or String.toLowerCase() in lambda$formatInterests$0(String)
```

Image 9. SpotBugs GUI subcategory internationalization containing 4 bugs.

The problem with the string conversion when dealing with international characters is shown in image 10.

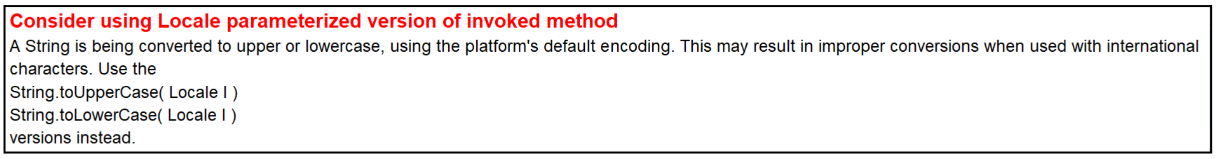

Consider using Locale parameterized version of invoked method
A String is being converted to upper or lowercase, using the platform's default encoding. This may result in improper conversions when used with international characters. Use the
String.toUpperCase(Locale l)
String.toLowerCase(Locale l)
versions instead.

Image 10. description of the issue with use of non-locale case conversion.

Malicious Code Vulnerability

The most prominent bug category was **malicious code vulnerability** with 70 bugs in total image 9. shows the types of subcategories it contains, which give a quick impression about the type of bug that is in question.

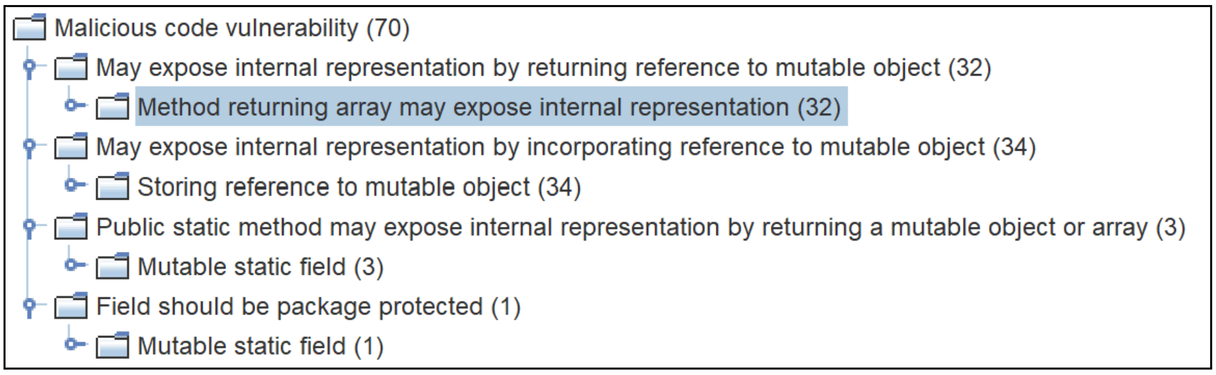
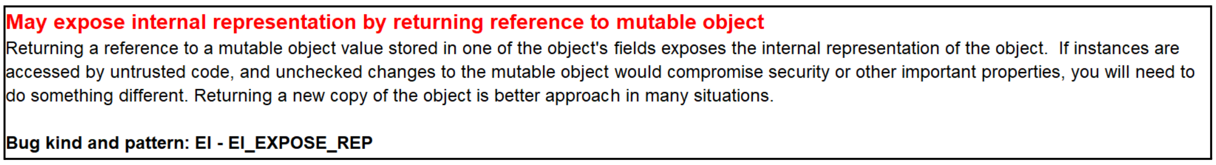

Malicious code vulnerability (70)
- May expose internal representation by returning reference to mutable object (32)
  - Method returning array may expose internal representation (32)
- May expose internal representation by incorporating reference to mutable object (34)
  - Storing reference to mutable object (34)
- Public static method may expose internal representation by returning a mutable object or array (3)
  - Mutable static field (3)
- Field should be package protected (1)
  - Mutable static field (1)

Image 9. SpotBugs GUI subcategory malicious code vulnerability containing 70 bugs.

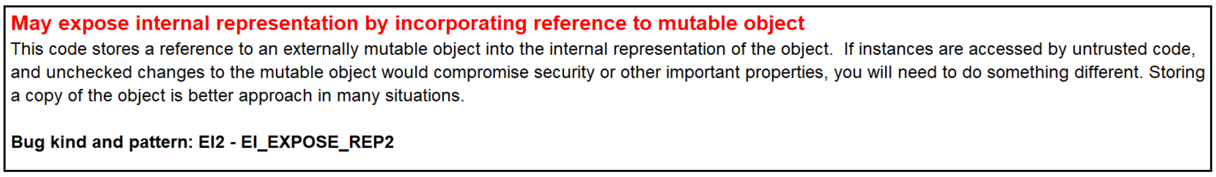
Most of the malicious code vulnerabilities are related to methods exposing internal representation by returning reference to mutable objects, which account for 32 bugs. GUI gives a more in depth description of the issue seen in image 10.


May expose internal representation by returning reference to mutable object
Returning a reference to a mutable object value stored in one of the object's fields exposes the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Returning a new copy of the object is better approach in many situations.

Bug kind and pattern: EI - EI_EXPOSE_REP

Image 10. description of the issue with returning a reference to a mutable object.

Another big contributor to the bugs is methods storing reference to mutable objects. The GUI gives a more in depth description of the issue of storing these references shown in image 11.


May expose internal representation by incorporating reference to mutable object
This code stores a reference to an externally mutable object into the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Storing a copy of the object is better approach in many situations.

Bug kind and pattern: EI2 - EI_EXPOSE_REP2

Image 11. description of the issue with storing a reference to a mutable object.

Mutable static fields account for the rest of the four bugs in the malicious code vulnerability. As seen in image 12.

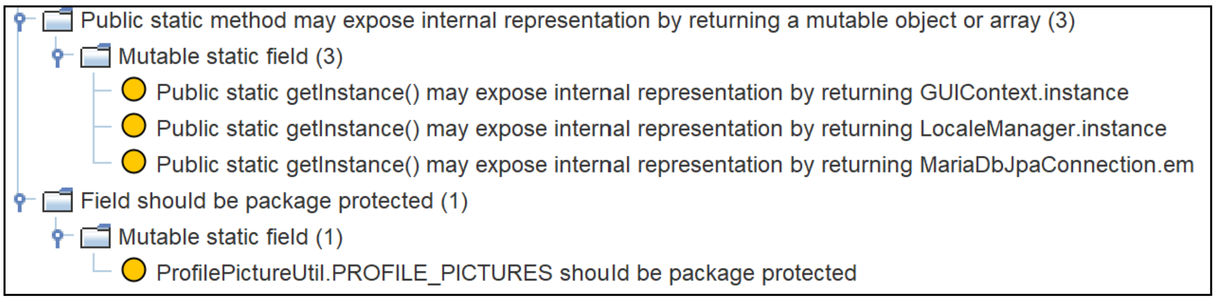
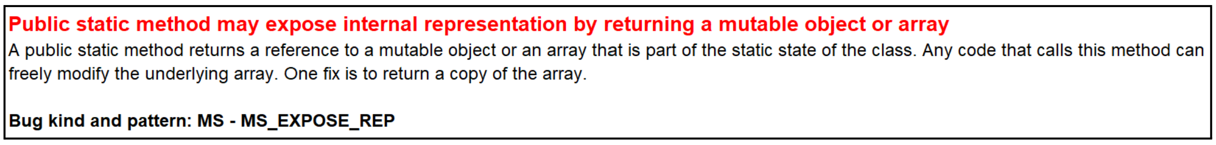

- Public static method may expose internal representation by returning a mutable object or array (3)
  - Mutable static field (3)
    - Public static getInstance() may expose internal representation by returning GUIContext.instance
    - Public static getInstance() may expose internal representation by returning LocaleManager.instance
    - Public static getInstance() may expose internal representation by returning MariaDbJpaConnection.em
- Field should be package protected (1)
  - Mutable static field (1)
    - ProfilePictureUtil.PROFILE_PICTURES should be package protected

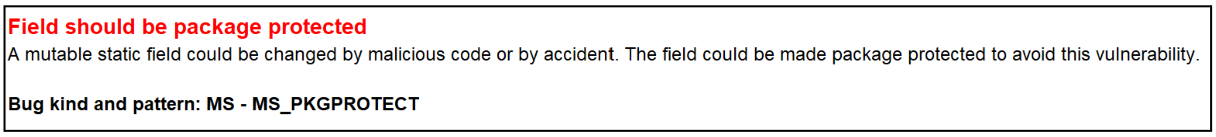
Image 12. shows the remaining categories under malicious code vulnerability.

Errors for these mutable static field bugs are as follows. In the first case, a public static method may expose internal representation by returning a mutable object or array seen in image 13. and in the second one a field should be package protected seen in image 14.


Public static method may expose internal representation by returning a mutable object or array
A public static method returns a reference to a mutable object or an array that is part of the static state of the class. Any code that calls this method can freely modify the underlying array. One fix is to return a copy of the array.

Bug kind and pattern: MS - MS_EXPOSE_REP

Image 13. description of the issue with a static method returning a mutable object or array.


Field should be package protected
A mutable static field could be changed by malicious code or by accident. The field could be made package protected to avoid this vulnerability.

Bug kind and pattern: MS - MS_PKGPROTECT

Image 14. description of the issue with a field not being package protected.

So most of the bugs in our project software has to do with mutability and how to handle it more effectively and securely. General solution for mutability issues would be using defensive copies, which means whenever storing or returning mutable objects to always use copies and not direct references. So cloning and using copies would be a much safer option leading to less vulnerability issues with malicious code. After this implementation and fixing of these bugs our program will be much more safe.

Multithreaded Correctness

The category **multithreaded correctness** had 2 bugs in total as seen in image 15. The bug are about thread safety. A static field is lazily initialized without volatile or any synchronization. Because of this other potential threads could see an invalid object. These bugs were both detected in MariaDbJpaConnection.getInstance(), where there is lazy initialization of static fields em and emf.

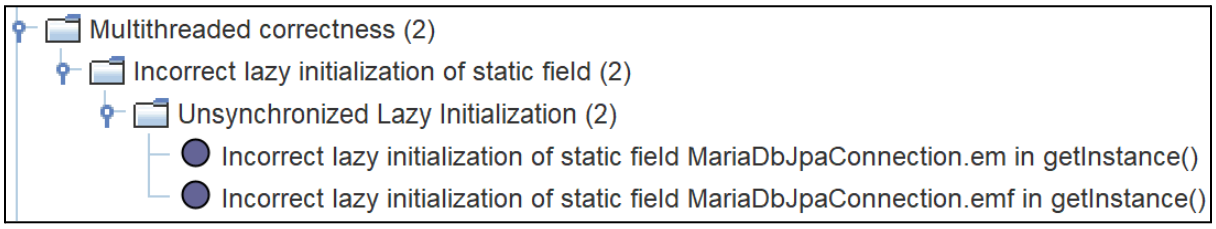
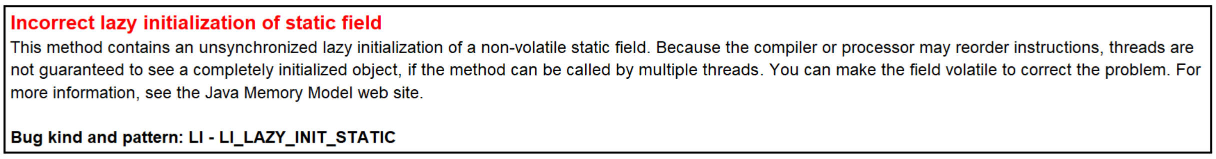

```
Multithreaded correctness (2)
    Incorrect lazy initialization of static field (2)
        Unsynchronized Lazy Initialization (2)
            ● Incorrect lazy initialization of static field MariaDbJpaConnection.em in getInstance()
            ● Incorrect lazy initialization of static field MariaDbJpaConnection.emf in getInstance()
```

Image 15. SpotBugs GUI subcategory multithreaded correctness containing 2 bugs

A more detailed description of the issue with incorrect lazy initialization of static fields is shown in the GUI as seen in image 16.

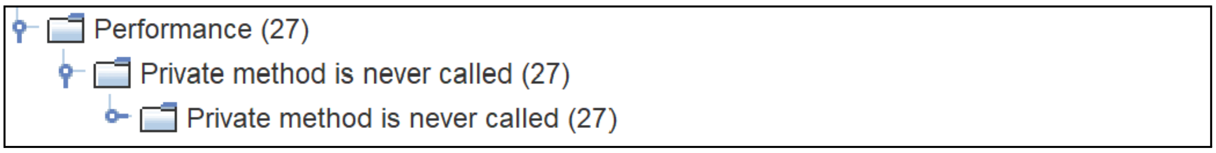

Incorrect lazy initialization of static field
This method contains an unsynchronized lazy initialization of a non-volatile static field. Because the compiler or processor may reorder instructions, threads are not guaranteed to see a completely initialized object, if the method can be called by multiple threads. You can make the field volatile to correct the problem. For more information, see the Java Memory Model web site.

Bug kind and pattern: LI - LI_LAZY_INIT_STATIC

Image 16. description of the issue with incorrect lazy initialization of static fields.

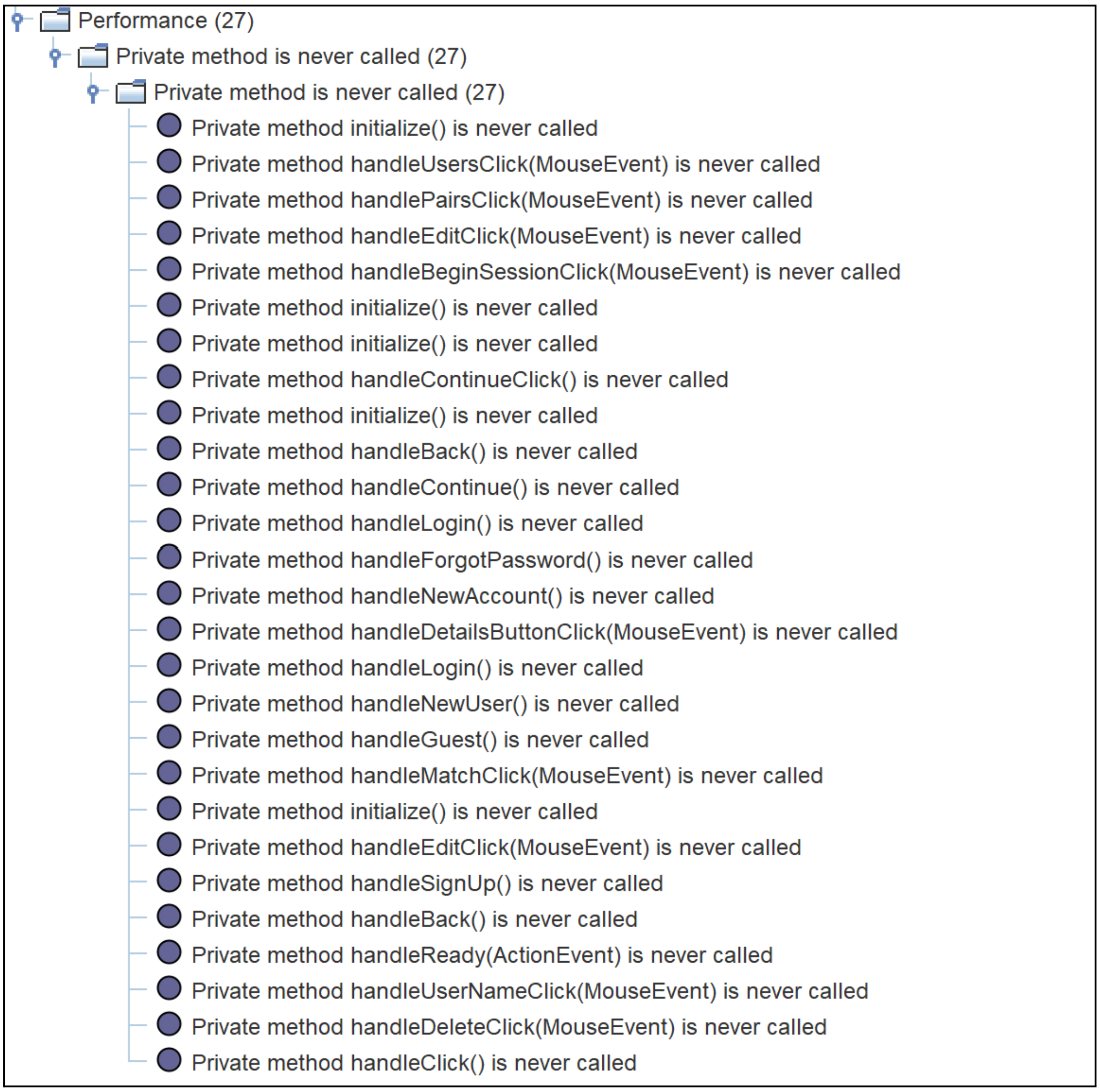
Performance

The subcategory **Performance** had 27 bugs seen in image 17. The performance bugs stem from unused private methods. Spotbugs claims the following methods in image 18. are never called from anywhere in the code. These methods appear unused in code but are triggered by FXML through JavaFX built in dependency injection and event handling. This happens in a method such as @FXML private void initialize() or @FXML private void handleLogin(ActionEvent e).



```
Performance (27)
  Private method is never called (27)
    Private method is never called (27)
```

Image 17. SpotBugs GUI subcategory performance containing 27 bugs.



```
Performance (27)
  Private method is never called (27)
    Private method is never called (27)
      Private method initialize() is never called
      Private method handleUsersClick(MouseEvent) is never called
      Private method handlePairsClick(MouseEvent) is never called
      Private method handleEditClick(MouseEvent) is never called
      Private method handleBeginSessionClick(MouseEvent) is never called
      Private method initialize() is never called
      Private method initialize() is never called
      Private method handleContinueClick() is never called
      Private method initialize() is never called
      Private method handleBack() is never called
      Private method handleContinue() is never called
      Private method handleLogin() is never called
      Private method handleForgotPassword() is never called
      Private method handleNewAccount() is never called
      Private method handleDetailsButtonClick(MouseEvent) is never called
      Private method handleLogin() is never called
      Private method handleNewUser() is never called
      Private method handleGuest() is never called
      Private method handleMatchClick(MouseEvent) is never called
      Private method initialize() is never called
      Private method handleEditClick(MouseEvent) is never called
      Private method handleSignUp() is never called
      Private method handleBack() is never called
      Private method handleReady(ActionEvent) is never called
      Private method handleUserNameClick(MouseEvent) is never called
      Private method handleDeleteClick(MouseEvent) is never called
      Private method handleClick() is never called
```

Image 18. shows all the methods that are claimed unused.

Discussion

To summarize SpotBugs findings and results. There were 115 bugs across 6 different categories in our project. Most of the bugs 70/115 were related to malicious code vulnerability, more specifically concerning mutability. The methods in question expose internal representations by returning or storing references to mutable objects and mutable static fields. To avoid this we can implement defensive copying. This way the internal state of a class cannot be modified by outside code, which will prevent unintended functionality and vulnerability.

The other bug categories present in the SpotBugs bug report were:

Correctness 1 bug: This bug was found in PairItemController.handleMatchClick(), where a null pointer dereference can potentially happen.

Bad practice 11 bugs: 8 of the 11 issues had to do with methods intentionally throwing RuntimeException. This was found in classes GuestDao, MatchDao, and UserDao. The final 3 issues were about the unsafe usage of getResource() in subclassess AdminHomeController, AdminUsersController, and view.GUI. The potential problems with resource loading arises when these classes are further extended.

Internationalization 4 bugs: These bugs were about using toUpperCase() or toLowerCase methods which can create incorrect results when using non-English languages. These bugs can be found in AfterMatchController, InterestSelectionController, ProfileController, and MatchUtils.

Multithreaded correctness 2 bugs: Both bugs are found in MariaDbJpaConnection.getInstance(), in which fields em and emf are initialized lazily without volatile or any synchronization. This could potentially lead to problems when other threads deal with just partially initialized objects.

Performance 27 bugs: Had to do with private methods that were being reported unused but are actually invoked via FXML with @FXML tag or in button handlers.

After these bugs are addressed the code will be more according to best practices and it will improve the maintainability and security of the code.