Maintainability Measurement Report on

Iluwatar – Java Design Patterns

Dr. Adam Vanya

FrontEndART Ltd.

12/14/2020

Table of Contents

Table of Figures

1 Introduction

1.1 The software analyzed

The Iluwatar – Java Design Patterns GitHub project (https://github.com/iluwatar) provides a codebase with Java design pattern implementations. The implementations have been developed by experienced programmers and architects from the open source community. The source code examples are intended to be programming tutorials on how to implement a specific pattern. Next to the implementations of the design patterns, this open source project also helps developers with textual descriptions of the patterns.

This open source GitHub project is highly popular with more than 62.000 stars (indication of how many people like this project), almost 20.000 forks (indication of how often the codebase was accessed) and more than 4000 watches (indication of how many people follow this project). In the codebase of this GitHub project currently 139 Java design patterns are implemented in total. From now on this codebase will be referred to as JDP.

1.2 Goal and motivation

A huge community is using JDP either to get inspirations for creating pattern implementations or to copy and reuse JDP's code directly. Therefore, JDP pattern implementations end up in many software systems. This implies a significant impact on the set of software that is being created around the world. Therefore, the risks of using JDP need to be identified and mitigated. Some of these risks are related to how maintainable the pattern implementations are.

By executing a **source code quality assessment** on JDP, FrontEndART Ltd. intends to help the community of JDP with identifying the maintainability related technical risks JDP has and with pieces of advice for mitigating those risks. These risks and pieces of advice can then be used to increase the maintainability of the pattern implementations and have a positive effect on the maintainability of all software systems using JDP. Next to that, FrontEndART Ltd. also finds this a great opportunity to showcase their source code quality assessment service. This report documents the results of this service.

1.3 Report structure

The rest of this report is structured as follows:

- **Section 2** details the scope of the analysis executed.
- **Section 3** describes the measurement results together with the detailed technical findings.
- **Section 4** summarizes the technical findings.
- **Section 5** suggests improvement steps to be taken.

2 Analysis scope

Version 2.2 of QualityGate was used to execute the analysis. This source code quality management system analyzed each of the main branch commits of the codebase and future ones are planned to be analyzed too. The subject of the current report is the commit which was checked-in on November 23, 2020 at 9:35 PM (SHA: 0c44b5390904f43afa34298b17093d65d63eb1cc). The scope of the inspection was maintainability assessment. During the assessment all Java files were analyzedincluding test, generated and 3rd party code. This is in line with how all the other systems in the FrontEndART benchmark were analyzed. Based on the analysis results, the expert assessment was concluded on 15th December 2020.

The components of the codebase are represented by the **concreteextensions**, **domainapp** and **com.iluwatar.*** packages. Based on this, the following 128 components were identified:

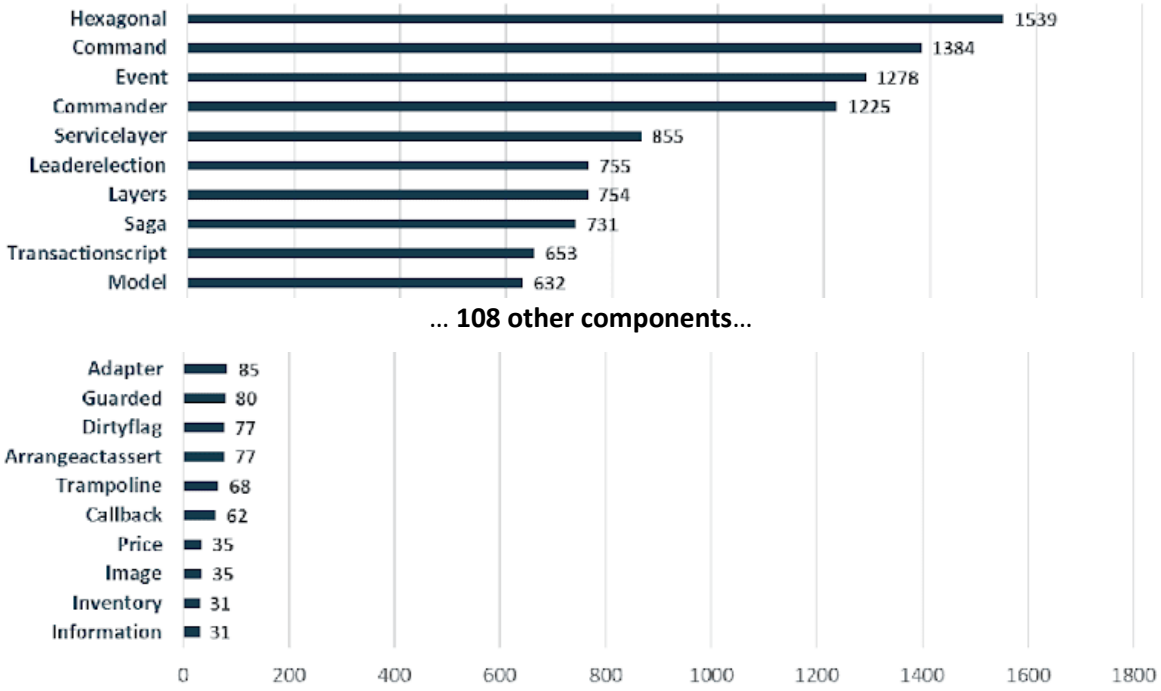

Figure 1*:* **Component size in terms of logical lines of code**

The following system-level measurements are characterizing the codebase analyzed:

System-level Metric	Value
Number of Components	**128**
Number of Methods	**4,359**
Logical Lines of Code	**37,768**
API Documentation Coverage	**54%**
Clone Coverage	**8%**

Clone Instances	**227**
Warnings – Blocker	**35**
Warnings – Critical	**272**
Warnings – Major	**864**
Warnings – Minor	**3,421**
Warnings – Info	**0**

3 Measurement results and technical findings

The maintainability of the Iluwatar – Java Design Patterns codebase is 7.94 on the scale from 0 (score of the least maintainable system) to 10 (score of the most maintainable system). This score means that the maintainability of the Iluwatar – Java Design Patterns codebase is OUTSTANDING considering the FrontEndART benchmark which includes hundreds of open source and closed source systems. The score of the snapshot analyzed may change somewhat over time since the benchmark changes too.

The following figure indicates the metrics and quality attributes based on which the maintainability score of the codebase was calculated:

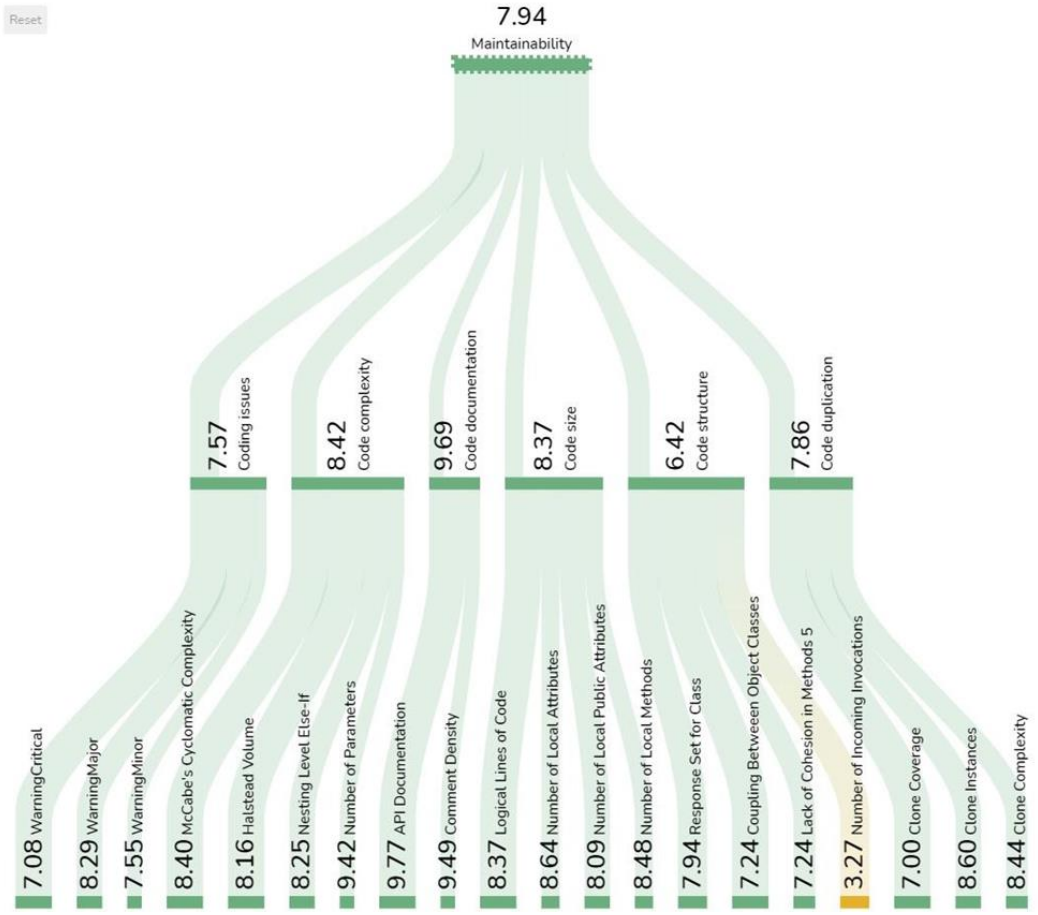

Figure 2*:* **FrontEndART Maintainability Model applied to Iluwatar – Java Design Patterns**

This section goes through the FrontEndART maintainability model described above and explains the elements of that model: 6 quality attributes and 20 metrics. For each model element this section also details the measurement results and the underlying technical findings for the Iluwatar – Java Design Patterns codebase. Those metrics where the Iluwatar – Java Design Patterns codebase performs less than OUTSTANDING are given extra attention when detailing measurement results.

3.1 Coding Issues

The Coding Issues quality attribute indicates the extent to which a given software system is implemented using programming best practices as compared to other systems in the FrontEndART benchmark. The Iluwatar – Java Design Patterns codebase scores 7.57 (GOOD) on this quality attribute. This codebase implements more best practices than most of the other systems in the benchmark.

3.1.1 WarningCritical, WarningMajor and WarningMinor

Coding violations are created by programmers when they do not follow programming best practices. In such cases, warnings can be created. Based on the effect of a violation on maintainability, one can distinguish between critical, major and minor warnings. The score of a warning metric is calculated by measuring the number coding violations for each class.

The codebase has in total 1,326 classes and these classes contain 272 critical violations, 864 major violations and 3,421 minor violations. Based on the measurements of each class and the FrontEndART benchmark, the WarningCritical, WarningMajor and WarningMinor metric scores are 7.08 (GOOD), 8.29 (OUTSTANDING) and 7.55 (GOOD), respectively.

A representative example of a critical violation can be found in the **DAO** component involving the dao/src/main/java/com/iluwatar/dao/**DbCustomerDao.java** file. In this file there is a method called **getAll()** which does not close a database connection. The following figure shows the content of this method:

```java
@Override 698~CloneInstance
public Stream<Customer> getAll() throws Exception { FH_TMR
  try {
    var connection = getConnection(); FH_CIR
    var statement = connection.prepareStatement("SELECT * FROM CUSTOMERS"); squid:S2095 FH_CIR
    var resultSet = statement.executeQuery(); // NOSONAR FH_CIR
    return StreamSupport.stream(new Spliterators.AbstractSpliterator<Customer>(Long.MAX_VALUE,
        Spliterator.ORDERED) {

      @Override
      public boolean tryAdvance(Consumer<? super Customer> action) { FH_TMR
        try {
          if (!resultSet.next()) {
            return false;
          }
          action.accept(createCustomer(resultSet));
          return true;
        } catch (SQLException e) {
          throw new RuntimeException(e); // NOSONAR FH_ATRET
        }
      }
    }, false).onClose(() -> mutedClose(connection, statement, resultSet));
  } catch (SQLException e) {
    throw new CustomException(e.getMessage(), e);
  }
}
```

Figure 3: **Critical violation example**

To decrease the negative effect of coding violations on maintainability the number of critical and minor violations should further be decreased by fixing those violations. The full list of the detected violations can be found on the website of the QualityGate system.

To help decide which components should be refactored first, the warning density for each component were calculated. The warning density is result of dividing the number of warnings in a component by the number of logical lines of code in the same component. This measurement was repeated for critical, major and minor warning separately.

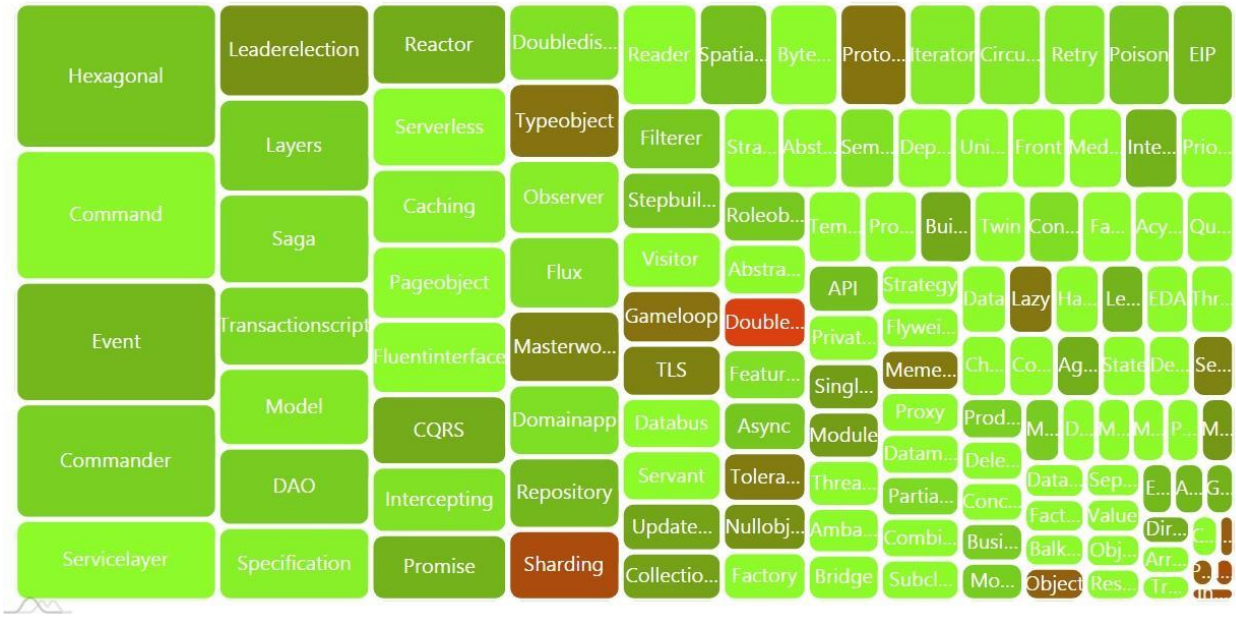

Figure 4 shows us that the **Doubledispatch**, **Typeobject** and **Prototype** components are not only relatively big as compared to other components, but they also have the highest critical warning density. One could consider addressing the critical warnings of these components first.

Figure 5: **Component level major warning density distribution**

Figure 5 indicates that from the biggest codebase components **Transactionscript**, **Commander** and **DAO** components have the highest major warning density. One could consider addressing the critical warnings of these components first.

Figure 6: **Component level minor warning density distribution**

Figure 6 points to **Async**, **Commander** and **Spatialpartition** as the components which are not only relatively big as compared to other components, but they also have the highest minor warning density. One could consider addressing the minor warnings of these components first.

3.2 Code Complexity

The Coding Complexity quality attribute indicates the extent to which a given software system is easy to understand, modify and test as compared to other systems in the FrontEndART benchmark. The Iluwatar–Java Design Patterns codebase scores 8.42 (OUTSTANDING) on this quality attribute. The code of this codebase is less complex than the code of most of the other systems in the benchmark.

3.2.1 McCabe's Cyclomatic Complexity

The McCabe's cyclomatic complexity metric counts all the possible execution paths for each methods of the system. This is achieved by counting the decision points in the code of those methods. A method with a high amount of execution paths is difficult to understand and test for functional suitability. For a maintainable system, the number of execution paths in a method must be kept minimal.

The codebase has in total 4,359 methods and the cyclomatic complexity of those methods reaches a maximum of 18 McCabe points. Based on the measurements of each method and the FrontEndART benchmark, the McCabe's Cyclomatic Complexity metric score is 8.16 (OUTSTANDING).

The methods of the codebase contain methods with a low McCabe complexity. This leads to methods that are easy to understand and test.

3.2.2 Halstead Volume

The Halstead volume metric evaluates the size of each method in terms of their length and vocabulary. Length and vocabulary are counted based on operators and operands. The methods with a high volume are difficult to read through, understand, test and reuse. Therefore, a maintainable system should contain methods with a low volume.

The codebase has in total 4,359 methods and the Halstead volume of those methods reaches a maximum of 5,082. Based on the measurements of each method and the FrontEndART benchmark, the Halstead Volume metric score is 8.16 (OUTSTANDING).

The methods of the codebase are short and use a limited vocabulary. This contributes to methods that are easy to understand and test.

3.2.3 Nested Level Else-If

This metric expresses the complexity of methods as the depth of the maximum embeddedness of its conditional and iteration block scopes, where in the if-else-if construct only the first if instruction is considered. A method with deeply embedded programming constructions is difficult to understand and test for functional suitability. For a maintainable system, the depth of embeddedness must be kept minimal.

The codebase has in total 4,359 methods and the maximum level of program construction embeddedness in those methods reaches a maximum of 5. Based on the measurements of each method and the FrontEndART benchmark, the Nested Level Else-If metric score is 8.25 (OUTSTANDING).

The methods of the codebase do not contain deep nesting of programming constructions, like loops and conditional statements. This contributes to methods that are easy to understand and test.

3.2.4 Number of Parameters

This metric measures how many parameters each method of the system has. Methods which have many parameters impose a risk of calling that method incorrectly and it may also indicate that the method implements more than just a single concept. Therefore, for a maintainable software one needs to keep the number of parameters in the signature of the methods low.

The codebase has in total 4,359 methods and the maximum number of parameters methods use is 12. Based on the measurements of each method and the FrontEndART benchmark, the Number of Parameters metric score is 9.42 (EXCEPTIONAL).

The methods of the codebase use just a few parameters in their signature. With this, calling a method is easy, less error prone and it is more likely that those methods implement just a single concept.

3.3 Code Documentation

The Coding Documentation quality attribute indicates the extent to which a given software system is documented as compared to other systems in the FrontEndART benchmark. The Iluwatar – Java Design Patterns codebase scores 9.69 (EXCEPTIONAL) on this quality attribute. This codebase belongs to one of the best documented systems in the benchmark.

3.3.1 API Documentation

This metric calculates the ratio of the number of documented public methods in the class (+1 if the class itself is documented) to the number of all public methods in the class (+ 1 the class itself); however, the nested, anonymous, and local classes are not included. The result of this calculation shows how well the public API of a class is documented. API documentation contributes a lot to understanding how that API should be used. Therefore, for an easy-to-understand and maintainable system the APIs must be properly documented.

The codebase has a total of 1,326 classes and the ratio of the documented public classes ranges between 0% and 100%. Based on the measurements of each class and the FrontEndART benchmark, the API Documentation metric score is 9.77 (EXCEPTIONAL).

Public methods are properly documented. This helps the users of the APIs understand how those APIs should be used properly.

3.3.2 Comment Density

This metric measures the ratio of the comment lines of the class to the sum of its comment and logical lines of code. In other words, the percentage of comment lines is calculated for each class. Documentation can help to make the code of a class more understandable and maintainable because it provides extra information. The importance of comments increases with the size of the class.

The codebase has 1,326 classes in total and the ratio of documented public classes ranges between 0% and 100%. Based on the measurements of each class and the FrontEndART benchmark, the Comment Density metric score is 9.77 (EXCEPTIONAL).

The classes are well documented with comments. These comments help to understand the necessary details of the implementation.

3.4 Code Size

The Code Size quality attribute indicates the extent to which a given software system is implemented using small and therefore easy-to-understand code elements as compared to other systems in the FrontEndART benchmark. The Iluwatar – Java Design Patterns codebase scores 8.37 (OUTSTANDING) on this quality attribute. This codebase in general has smaller code elements than most of the other systems in the benchmark.

3.4.1 Logical Lines of Code

This metric measures the size of the methods by counting the number of non-empty and non-comment code lines of the method; however, its anonymous and local classes are not included. Long methods are more difficult to understand and they often contain implementation of multiple concepts. Such contained concept implementations cannot be reused. For sound maintainability the size of the system methods must be kept small.

The codebase has in total 4,359 methods and the maximum number non-empty and non-comment lines in the methods is 96. Based on the measurements of each method and the FrontEndART benchmark, the Logical Lines of Code metric score is 8.37 (OUTSTANDING).

The methods of the codebase contain a low number of lines of code with are not empty and not comment lines. Small methods are easier to read through, understand, test and reuse.

3.4.2 Number of Local Attributes

This metric measures the amount of responsibility each class has by counting the number of local attributes for those classes. The more local attributes a class has, the more difficult concept that class implements. It can also indicate that the class implements multiple concepts. Classes with difficult concept implementations are difficult-to-understand and test. Therefore, from a maintainability point of view, the number of local attributes must be kept low.

The codebase has in total 1,326 classes and the maximum number of local attributes within a class is 15. Based on the measurements of each class and the FrontEndART benchmark, the Number of Local Attributes metric score is 8.64 (OUTSTANDING).

The number of local attributes in the classes of the codebase are low. This indicates that the responsibility of those classes is likely to be low as well and that the separation of concepts is correctly implemented.

3.4.3 Number of Local Public Attributes

This metric measures how well one of the fundamental concepts of objected-oriented programming, namely encapsulation, is implemented by a class. The more local public attributes a class has, the more open the implementation of that class is and the less information hiding, or encapsulation is achieved. Open class implementations can quickly lead to unwanted couplings and code that is difficult to understand and test. To avoid this, the number of public local class attributes must be kept low.

The codebase has in total 1,326 classes and the maximum number of local attributes within a class is 5. Based on the measurements of each class and the FrontEndART benchmark, the Number of Local Public Attributes metric score is 8.09 (OUTSTANDING).

The number of local public attributes in the classes of the coedbase are low. This indicates that the encapsulation within the classes is correctly implemented.

3.4.4 Number of Local Methods

This metric inspects the diversity of a class's behavior by counting the number of methods of that class. Classes with many methods are difficult to understand and to identify what exactly that class is responsible for. In such cases it is wise to re-think what that class should do and execute the necessary changes.

The codebase has in total 1,326 classes and the maximum number for local methods within a class is 18. Based on the measurements of each class and the FrontEndART benchmark, the Number of Local Methods metric score is 8.48 (OUTSTANDING).

The number of local methods in the classes of the codebase are low. This indicates that the responsibility of those classes is likely to be low as well and that the separation of concepts is correctly implemented.

3.5 Code Structure

The Code Structure quality attribute indicates the extent to which a given software system is structured as compared to other systems in the FrontEndART benchmark. The Iluwatar – Java Design Patterns codebase scores 6.42 (ABOVE MARKET AVERAGE) on this quality attribute. This codebase is structured better than an average system in the benchmark.

3.5.1 Response Set for Class

This metric measures the testability and analyzability of classes by counting the sum of the methods to which that class refers to. The more methods supply data to a class, the more difficult it is to test and analyze that class. To implement a maintainable system, the number of referred methods must be kept low.

The codebase has in total 1,326 classes and the maximum response set size for a class is 30. Based on the measurements of each class and the FrontEndART benchmark, the Response Set for Class metric score is 7.94 (OUTSTANDING).

The amount of methods called by the classes of the codebase are typically low. This means that those classes are typically compact, and their lifecycle isless likely to be connected to those of other classes.

3.5.2 Coupling Between Object Classes

This metric measures the size of the environment of a class by counting the number of referred classes. These references can be related to inheritance, method call, type reference, etc. Classes which depend on many other classes are more difficult to test and to reuse. Moreover, these classes are overly sensitive to the changes in their environment. It is therefore needed to keep weak couplings between classes.

The codebase has in total 1,326 classes and the maximum the classes a class uses is 16. Based on the measurements of each class and the FrontEndART benchmark, the Coupling Between Object Classes metric score is 7.24 (GOOD).

An example of a class with a high number of used classes can be found in the Specification component involving the specification/src/main/java/com/iluwatar/specification/app/App.java file. The App class in this file uses 16 other classes. The following figure shows some of the class references from this class:

```java
public class App { PMD SCN

  private static final Logger LOGGER = LoggerFactory.getLogger(App.class);

  /**
   * Program entry point.
   */
  public static void main(String[] args) {
    // initialize creatures list
    var creatures = List.of(
        new Goblin(),
        new Octopus(),
        new Dragon(),
        new Shark(),
        new Troll(),
        new KillerBee()
    );
    // so-called "hard-coded" specification
    LOGGER.info("Demonstrating hard-coded specification :");
    // find all walking creatures
    LOGGER.info("Find all walking creatures");
    print(creatures, new MovementSelector(Movement.WALKING));
    // find all dark creatures
    LOGGER.info("Find all dark creatures");
    print(creatures, new ColorSelector(Color.DARK));
    LOGGER.info("\n");
```

Figure 7: **Example of a class coupled to 16 other classes**

To decrease the negative effect of class couplings on maintainability the number of used classes must be decreased. The full list of classes with many classes used can be found on the website of the QualityGate. From these the following list contains the ones with the most classes used.

Class	Number of class relations
…/iluwatar/specification/app/App.java : **App**	16
…/iluwatar/commander/AppShippingFailCases.java : **AppShippingFailCases**	15
…/iluwatar/commander/Commander.java : **Commander**	15
…/iluwatar/commander/AppEmployeeDbFailCases.java : **AppEmployeeDbFailCases**	14
…/iluwatar/commander/AppPaymentFailCases.java : **AppPaymentFailCases**	14
…/iluwatar/commander/AppQueueFailCases.java : **AppQueueFailCases**	14
…/iluwatar/commander/AppMessagingFailCases.java : **AppMessagingFailCases**	13
…/iluwatar/hexagonal/domain/LotteryAdministration.java : **LotteryAdministration**	11
…/iluwatar/layers/service/CakeBakingServiceImpl.java : **CakeBakingServiceImpl**	11
…/iluwatar/prototype/App.java : **App**	10

Figure 8: **Top 10 classes with the highest number of class couplings**

In order to decide the issues of which component should be address first, Figure 9 indicates for each component the maximum number of coupled classes, the issue density and the number of issues within that component. Issue density is the ratio of the number of issues and the number of logical lines of code within a component. To reduce clutter, only those components were included where that maximum number of coupled classes is more than 66% of the codebase level maximum.

Figure 9: **Components with the most severe issues**

The Commander component has the highest issue density and it has 6 classes that are coupled up to 15 other classes one by one. Considering the number of class couplings, only the Specification component performs worse: it has a single class that is coupled to 16 other classes.

3.5.3 Lack of Cohesion in Methods 5

This metric is used to identify the number of functionalities of a class. One of the basic principles of object-oriented programming is encapsulation, meaning that attributes belonging together and the operations that use them should be organized into one class, and one class shall implement only one functionality, i.e. its attributes and methods should be coherent.

This metric measures the lack of cohesion and computes into how many coherent classes the class could be split. It is calculated by taking a non-directed graph, where the nodes are the implemented local methods of the class and there is an edge between the two nodes if and only if a common (local or inherited) attribute or abstract method is used or a method invokes another. The value of the metric is the number of connected components in the graph not counting those, which contain only constructors, destructors, getters, or setters. A maintainable system should contain classes with a single functionality.

The codebase has a total of 1,326 classes and the maximum number for local attributes within a class is 6. Based on the measurements of each class and the FrontEndART benchmark, the Lack of Cohesion in Methods 5 metric score is 7.24 (GOOD).

An example of a class which can be decomposed into multiple other classes can be found in the Caching component involving the caching/src/main/java/com/iluwatar/caching/AppManager.java file. The AppManager class in this file can be decomposed into 4 other classes. The following figure shows some of the class references from this class:

```
36  public final class AppManager {
37
38    private static CachingPolicy cachingPolicy;
39
40    private AppManager() {
41    }
42
43    /**
44     * Developer/Tester is able to choose whether the application should use MongoDB as its underlying
45     * data storage or a simple Java data structure to (temporarily) store the data/objects during
46     * runtime.
47     */
48    public static void initDb(boolean useMongoDb) {
49      if (useMongoDb) {
50        try {
51          DbManager.connect();
52        } catch (ParseException e) {
53          e.printStackTrace(); FH_APST squid:S1148
54        }
55      } else {
56        DbManager.createVirtualDb();
57      }
58    }
59
60    /**
61     * Initialize caching policy.
62     */
63    public static void initCachingPolicy(CachingPolicy policy) {
64      cachingPolicy = policy; FH_NFSVMBSB
65      if (cachingPolicy == CachingPolicy.BEHIND) { FH_NFSVMBSB
66        Runtime.getRuntime().addShutdownHook(new Thread(CacheStore::flushCache));
67      }
68      CacheStore.clearCache();
69    }
```

Figure 10: **Example of a class that can be decomposed based on the functionalities implemented**

To decrease the negative effect of classes with weak cohesion on maintainability the problematic classes need to be split up. The full list of classes without a sufficient level of cohesion can be found on the website of the QualityGate. From these the following list contains the ones with the weakest cohesion.

Class	Number of functionalities
.../iluwatar/reader/writer/lock/ReaderWriterLock.java : **WriteLock**	6
.../iluwatar/objectmother/RoyaltyObjectMother.java : **RoyaltyObjectMother**	6
.../iluwatar/reader/writer/lock/ReaderWriterLock.java : **ReadLock**	6
.../iluwatar/fluentinterface/fluentiterable/lazy/LazyFluentIterable.java : **LazyFluentIterable**	5
.../iluwatar/stepbuilder/CharacterStepBuilder.java : **CharacterSteps**	5
.../iluwatar/model/view/presenter/FileSelectorJFrame.java : **FileSelectorJFrame**	5
.../iluwatar/builder/Hero.java : **Builder**	5
.../domainapp/webapp/SimpleApplication.java : **SimpleApplication**	4
.../iluwatar/leaderelection/ring/RingInstance.java : **RingInstance**	4
.../iluwatar/servant/Servant.java : **Servant**	4

Figure 11: **Top 10 classes with the highest number of functionalities implemented**

In order to decide the issues of which component should be address first, Figure 12 indicates for each component the maximum number of functionalities in the classes, the issue density and the number of issues within that component. Issue density is the ratio of the number of issues and the number of logical lines of code within a component. To reduce clutter, only those components were included where that maximum number of functionalities in the classes is more than 66% of the codebase level maximum.

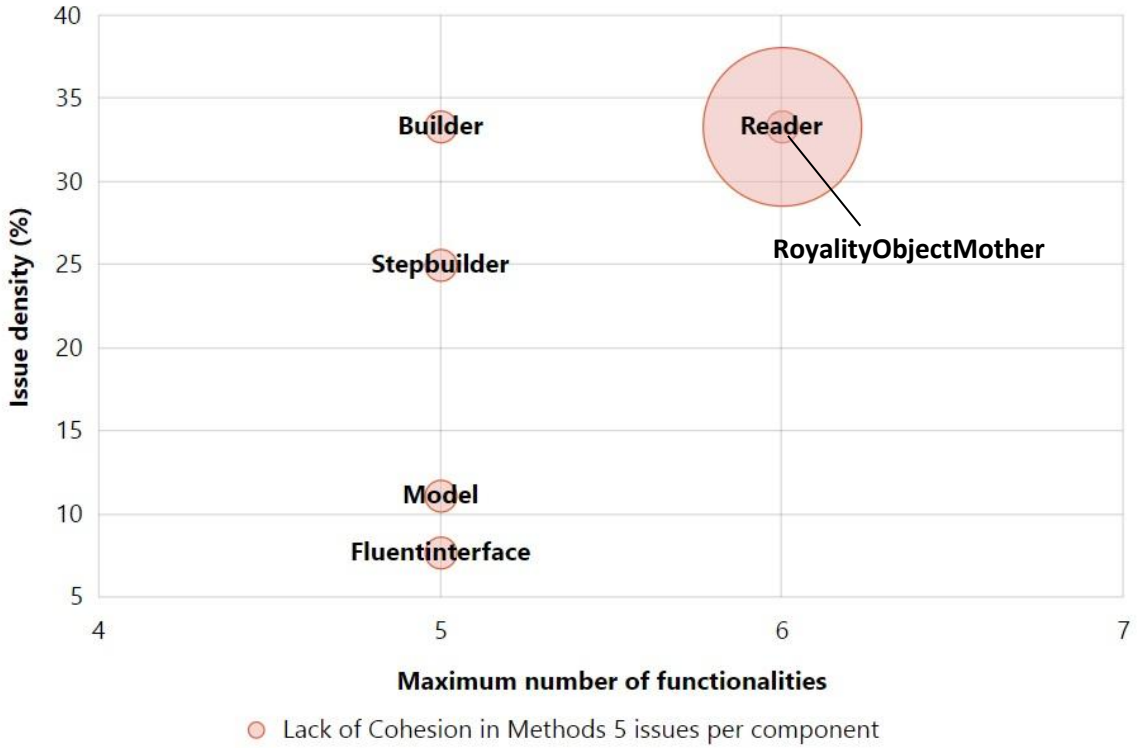

Figure 12: **Components with the most severe issues**

The Reader, RoyaltyObjectMother, Builder, Stepbuilder, Model and Fluentinterface components have classes that implement more than 4 functionalities in the whole codebase. From these components the Reader, RoyalityMother and Builder components are the ones where more than 30% of the classes implement more than 4 functionalities. Therefore, the issues within these components are good candidates for a first-round refactoring.

3.5.4 Number of Incoming Invocations

This metric measures the number of other methods and attribute initializations which directly call the local methods of a class. If a method is invoked several times from the same method or attribute initialization, it is counted only once. The more call a class has to its methods, the bigger the calling environment of that class is. Changes to a class with a big calling environment has risks, because they may imply changes to a high number of other classes. Those changes are typically difficult to perform and may lead to inconsistencies and bugs. An easily maintainable software has a low amount of incoming invocations for its classes, especially for those that are bigger in size.

The codebase has in total 1,326 classes and the maximum number of incoming invocations for a class is 30. Based on the measurements of each class and the FrontEndART benchmark, the Number of Incoming Invocations metric score is 3.27 (BELOW MARKET AVERAGE).

An example of a class which has many invocations to its local methods can be found in the Transactionscript component involving the transaction-script/src/main/java/com/iluwatar/transaction-script/HotelDaoImpl.java file. The local methods of the HotelDaoImpl class in this file are called 21 times. The following figure shows the first couple of lines of this class's implementation:

```
39  public class HotelDaoImpl implements HotelDao {
40    private static final Logger LOGGER = LoggerFactory.getLogger(App.class); FH_UPF
41
42    private final DataSource dataSource;
43
44    public HotelDaoImpl(DataSource dataSource) {
45      this.dataSource = dataSource;
46    }
47
48    @Override 702~CloneInstance
49    public Stream<Room> getAll() throws Exception { FH_TMR
50      try {
51        var connection = getConnection(); FH_CIR
52        var statement = connection.prepareStatement("SELECT * FROM ROOMS"); squid:S2095 FH_CIR
53        var resultSet = statement.executeQuery(); // NOSONAR FH_CIR
54        return StreamSupport.stream(new Spliterators.AbstractSpliterator<Room>(Long.MAX_VALUE,
55            Spliterator.ORDERED) {
```

Figure 13: **Example of a class with a high number of incoming invocations**

To decrease the negative effect of classes with many incoming invocations on maintainability, the problematic classes need to be refactored, for example by splitting them up. The full list of classes with a high amount of incoming invocations can be found on the website of the QualityGate. From these the following list contains the ones with the most incoming invocations.

Class	Number of incoming invocations
.../iluwatar/dao/Customer.java : **Customer**	30
.../iluwatar/transactionscript/Room.java : **Room**	25
.../iluwatar/hexagonal/domain/PlayerDetails.java : **PlayerDetails**	23
.../iluwatar/leaderelection/Message.java : **Message**	22
.../iluwatar/transactionscript/HotelDaoImpl.java : **HotelDaoImpl**	21
.../iluwatar/hexagonal/domain/LotteryNumbers.java : **LotteryNumbers**	19
.../commander/messagingservice/MessagingService.java : **MessagingService**	19
.../iluwatar/commander/User.java : **User**	18
.../iluwatar/commander/queue/QueueDatabase.java : **QueueDatabase**	18
.../iluwatar/doubledispatch/GameObject.java : **GameObject**	18

Figure 14: **Top 10 classes with the highest number of incoming invocations**

In order to decide the issues of which component should be address first, Figure 15 indicates for each component the maximum number of incoming invocations for the classes, the issue density and the number of issues within that component. Issue density is the ratio of the number of issues and the number of logical lines of code within a component. To reduce clutter, only those components were included where that maximum number of incoming invocations in the classes is more than 66% of the codebase level maximum.

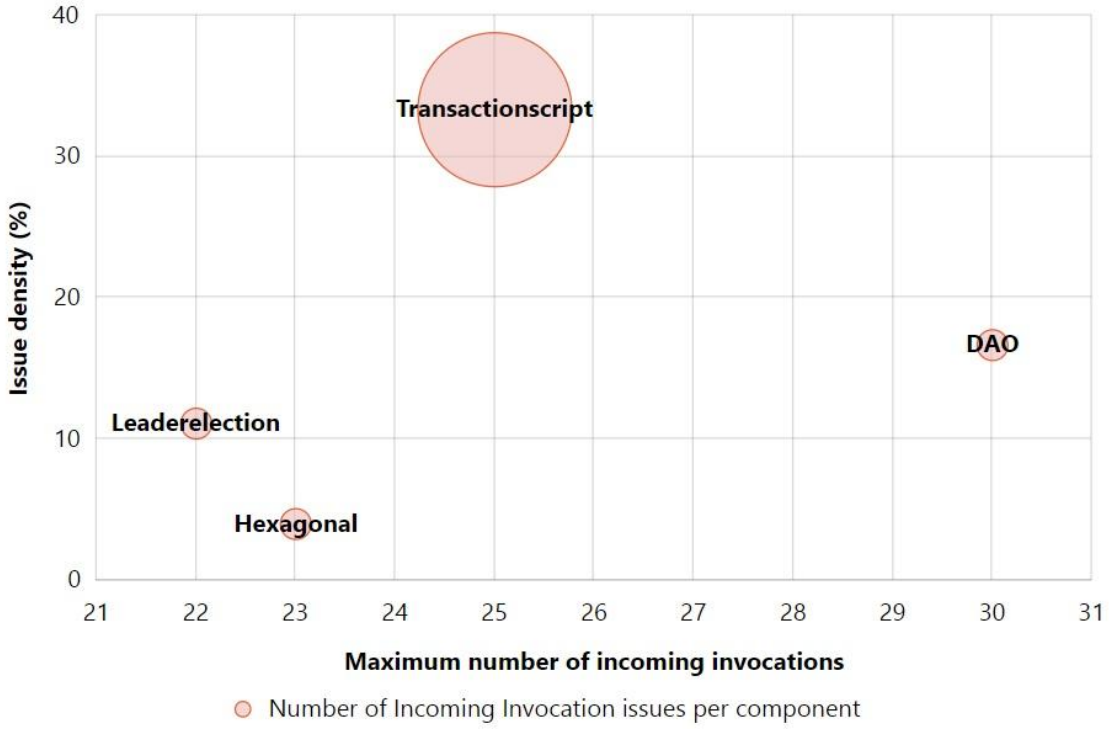

Figure 15: **Components with the most severe issues**

The Transcript, DAO, Leaderelection and Hexagonal components have classes with more than 21 incoming invocations. In the Transactionscript component these classes form 33% of the total number of classes. With this, Transactionscript has the highest issue density. Although the DAO component has a lower issue density, it has also a class where the number of incoming invocations is 30. These components need refactoring the most.

3.6 Code Duplication

The Code Duplication quality attribute indicates the extent to which the code of a given software system is duplicated as compared to other systems in the FrontEndART benchmark. The Iluwatar – Java Design Patterns codebase scores 7.86 (GOOD) on this quality attribute. The duplication related technical risks for this codebase is less severe than those for most of the other systems in the benchmark.

3.6.1 Clone Coverage

Two code segments are considered to be duplicates of each other when their code structures are the same (type 2 clones). The clone coverage metric measures the ratio of duplicated code within each class. To determine those ratios, syntactic entities (statements, expressions, etc.) are counted. The higher the clone coverage is within a class; the more code must be maintained in a consistent way. Maintaining duplicated code is error prone due to the likely inconsistent changes. It takes also more time and leads to an unnecessary increase of system size. Furthermore, duplicated code cannot be reused unless the entire class or method is a duplicate.

The codebase has in total 1,326 classes and the ratio of duplicated code in classes goes up to 100% in some cases. Based on the measurements of each class and the FrontEndART benchmark, the Clone Coverage metric score is 7.00 (GOOD).

An example of a class with a high clone coverage ratio can be found in the **CQRS** component involving the cqrs/src/main/java/com/iluwatar/cqrs/commandes/**CommandServiceImpl.java** file. In this file **69%** of the implementation of the **CommandSeviceImpl class** is covered by duplicates. The following figure shows two methods from this class:

```
private Author getAuthorByUsername(String username) { 353~CloneInstance
  Author author;
  try (var session = sessionFactory.openSession()) {
    var query = session.createQuery("from Author where username=:username");
    query.setParameter("username", username);
    author = (Author) query.uniqueResult();
  }
  if (author == null) {
    HibernateUtil.getSessionFactory().close();
    throw new NullPointerException("Author " + username + " doesn't exist!"); FH ATNPE
  }
  return author;
}

private Book getBookByTitle(String title) { 354~CloneInstance
  Book book;
  try (var session = sessionFactory.openSession()) {
    var query = session.createQuery("from Book where title=:title");
    query.setParameter("title", title);
    book = (Book) query.uniqueResult();
  }
  if (book == null) {
    HibernateUtil.getSessionFactory().close();
    throw new NullPointerException("Book " + title + " doesn't exist!"); FH ATNPE
  }
  return book;
}
```

Figure 16: **Example of a class with a high percentage of clone coverage**

The above two methods are duplicates of each-other. By relying on and using the fundamental concepts of object-orientation (for example: inheritance) one could reduce the redundancy introduced to a point where only a single method is defined and reused. The full list of the detected class-level duplications can be found on the website of the QualityGate. From these the following list contains the ones with the highest clone coverage.

Class	Clone Coverage
.../iluwatar/dao/DbCustomerDao.java : **DbCustomerDao**	100%
.../iluwatar/transactionscript/HotelDaoImpl.java : **HotelDaoImpl**	100%
.../iluwatar/leaderelection/bully/BullyApp.java : **BullyApp**	99%
.../iluwatar/leaderelection/ring/RingApp.java : **RingApp**	99%
.../iluwatar/intercepting/filter/DepositFilter.java : **DepositFilter**	94%
.../iluwatar/intercepting/filter/OrderFilter.java : **OrderFilter**	94%
.../iluwatar/servicelayer/spellbook/SpellbookDaoImpl.java : **SpellbookDaoImpl**	94%
.../iluwatar/servicelayer/wizard/WizardDaoImpl.java : **WizardDaoImpl**	94%
.../iluwatar/servicelayer/spell/SpellDaoImpl.java : **SpellDaoImpl**	94%
.../iluwatar/acyclicvisitor/App.java : **App**	90%

Figure 17: **Top 10 classes with the higest percentage of clone coverage**

In order to decide the issues of which component should be address first, Figure 18 indicates for each component the maximum percentage of clone coverage for the classes, the issue density and the number of issues within that component. Issue density is the ratio of the number of issues and the number of logical lines of code within a component. To reduce clutter, only those components were included where that maximum percentage of clone coverage in the classes is more than 66% of the codebase level maximum.

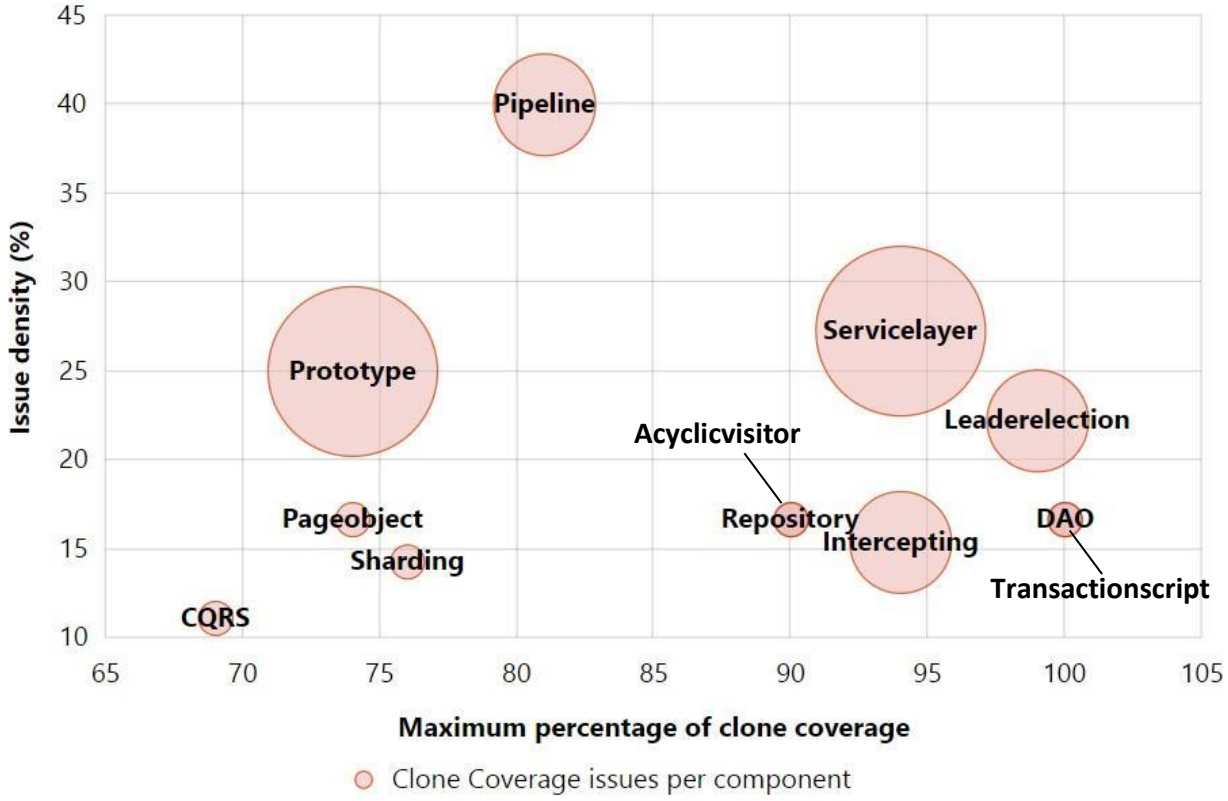

Figure 18**: Components with the most severe issues**

On the one hand, the Pipeline component has the highest percentage (40%) of its classes with high number of incoming invocations. On the other hand, there are other components with classes having even more incoming invocations. These are the DAO, Transactionscript, Leaderelection, Servicelayer, Intercepting, Repository and Acyclicvisitor components. These components should be refactored first if one wants to solve the most critical issues about incoming invocations.

3.6.2 Clone Instances

This metric measures the number of clone instances for each class of the system. Each clone instance is a piece of code that can be found somewhere else in the system with the same structure (type 2 duplicates). Clone instances can exist in the same class too. The more clone instances there are in a class, the higher

the chance is that changing another class with the same clone instances will lead to a change to that class. In other words, high number of clone instances in a class results in strong couplings between the class and its environment. Since there are no evident signs of these couplings, maintaining them are risky and can cause inconsistent changes.

The codebase has in total 1,326 classes and the maximum number of clone instances within a class is 6. Based on the measurements of each class and the FrontEndART benchmark, the Clone Instances metric score is 8.60 (OUTSTANDING).

The number of clone instances in the classes of the codebase are low. This is an indication that common code segments are properly separated and reused.

3.6.3 Clone Complexity

The clone complexity metric measures how complex the implementation of a class is that belongs to the clone instances of that class. This is done by taking the sum of McCabe complexity of the clone instances in the class. Complex clones are difficult to understand and to refactor. High complexity clones also increase the chance that the related clone instances will not be changed in a consistent way and that therefore a bug will be introduced.

The codebase has in total 1,326 classes and the maximum level of clone complexity within a class is 24 McCabe points. Based on the measurements of each class and the FrontEndART benchmark, the Clone Complexity metric score is 8.44 (OUTSTANDING).

Even in those classes where clone instances are present, the complexity of the code within those clone instances are low. Therefore, the effects of the existing clone instances are less severe.

4 Technical Summary

The maintainability of the Iluwatar – Java Design Patterns codebase is **7.94** on the scale from 0 (score of the least maintainable system) to 10 (score of the most maintainable system). This score means that the maintainability of the Iluwatar – Java Design Patterns codebase is **OUTSTANDING** considering the FrontEndArt benchmark which includes hundreds of open source and closed source systems.

Considering the 6 quality attributes of the FrontEndART Maintainability Model, the Iluwatar – java Design Patterns codebase performs the worst on the Coding Structure quality attribute with a score of **6.42**. However, even this score the codebase's code is better structured than an average system from the FrontEndART benchmark.

At the level of metrics, there are 6 metrics where the Iluwatar – Java Design Patterns codebase performs less than outstanding as compared to the FrontEndART banchmark. These are the following:

Metric	Score
Number of Incoming Invocations	**3.27**
Clone Coverage	**7.00**
WarningCritical	**7.08**
Coupling Between Object Classes	**7.24**
Lack of Cohesion in Methods 5	**7.24**
WarningMinor	**7.55**

This means that in the codebase analyzed, there are still a couple of classes with a high number of incoming invocations, high percentage of clone coverage, several warnings, many class couplings, and multiple functionalities implemented. Where to look for these classes are described in Sections 4.1.1, 4.5.2, 4.5.3, 4.5.4 and 4.6.1.

All the other 14 metrics indicate that from their perspective the maintainability of the Iluwatar – Java Design Patterns codebase is outstanding or even better.

5 Technical Advice

To improve the maintainability of the Iluwatar – Java Design Patterns codebase even further we advise addressing those issues which belong to the metrics with a less than outstanding score. These metrics are enumerated in Section 5. Based on this, FrontEndART provides the following piece of advice:

- Decrease the number of incoming invocations of the classes from the Transcript, DAO, Leaderelection and Hexagonal components. For example, by redesigning class responsibilities and splitting up class implementations.
- Decrease the clone coverage percentage of the classes from the Pipeline, DAO, Transactionscript, Leaderelection, Servicelayer, Intercepting, Repository and Acyclicvisitor components by factoring out and reusing duplicated code segments.
- Resolve critical coding violations in the Doubledispatch, Typeobject and Prototype components.
- Decrease the number of classes couplings related to the classes from the Commander, Specification, Hexagonal and Layers components. For example, by redesigning class responsibilities and splitting up class implementations.
- Decrease the number of functionalities classes implement from the Reader, RoyaltyObject-Mother, Builder, Stepbuilder, Model and Fluentinterface components by splitting them up where appropriate.
- Resolve minor coding violations in the Async, Commander and Spatialpartition components.

To make sure that the maintainability improves as required and stays at an acceptable level, we suggest monitoring the Iluwatar – Java Design Patterns codebase continuously. Such a monitoring service is

present within the service portfolio of FrontEndART Ltd. and we look forward to helping you manage maintainability on the long run.