3 Prioritization of Code Anomalies

By implementing a mechanism for detecting architecturally relevant code anomalies, we are already able to outline to developers which anomalies should be dealt with first. However, such mechanism is still insufficient for helping developers prioritize their refactorings for three main reasons: first, it could still result in unmanageable lists of anomalies, especially in large systems. Second, architecture problems are usually related to multiple occurrences of code anomalies. Therefore, in order to solve an architecture problem, developers should be able to identify and remove all of the anomalies causing it. Third, analyzing code anomaly patterns as indicators of architecture problems might produce many false positives. For example, classes that implement the Façade pattern might be classified as God Classes, even though they are not architecturally relevant – as they could be naturally related to several concerns. When analyzing the PDP system (MACIA *et al.*, 2012b), we detected a God Class infected with both Large Class and Shotgun Surgery anomalies (FOWLER *et al.*, 1999); this co-occurrence pattern is known as a good indicator of architecture problems, but in this case, developers did not consider this class architecturally relevant because it was fairly stable. That shows us that there might be other characteristics (in this case, number of changes) that affect the relevance of code anomalies. For this particular case, if we combine historic information regarding how many times this class was changed, we might improve the prioritization effectiveness.

In order to help developers in identifying and ranking architecturally relevant code anomalies, we propose a series of different prioritization heuristics. Those heuristics are based on four characteristics – or factors – that could indicate how critical an anomaly is with respect to architecture degradation. Thus, those heuristics analyze the code those relevant anomalies infect, in order to prioritize better candidates for refactoring. In this context, the prioritization mechanism works by granting scores to each code anomaly, according to the heuristic under analysis; using the example above, the detected God Class would have a low score under the change-proneness heuristic, decreasing its position on the final prioritization ranking.

3.1. Prioritization Heuristics

Many factors could indicate how harmful an anomaly really is to the system's architecture. These factors are exploited by the prioritization heuristics we propose. The choice of the factors was based on the definition of a set of desirable properties each of them should have. Therefore, our prioritization heuristics had to adhere to the following criteria:

1) *Relation to architecture problems*: previous studies in the literature have investigated the relation between the candidate characteristic and architecture problems. For example, the manifestation of errors on systems with architecture problems has been investigated by Weißgerber and Diehl (2006). Thus, the number of errors observed on a system is a good candidate characteristic for prioritization heuristics.
2) *Information availability*: in order to evaluate the prioritization heuristics, the characteristics they explore should be available on most systems we analyzed, regardless of their architecture designs.
3) *Automation*: the heuristics should have an algorithmic definition, in order to be eligible for automation. Therefore, only heuristics that could be automated should be considered.

We defined four prioritization heuristics, according to the aforementioned criteria. Those heuristics are described next.

3.1.1. Change-proneness Heuristic

When a code element suffers multiple changes through the systems' evolution, we call it *unstable* (KELLY, 2006). This heuristic is based on the idea that anomalies infecting unstable code elements are more likely to be architecturally relevant.

Definition. This heuristic calculates its ranking results according to the number of changes made to anomalous code elements. In this sense, given a code element *c,* this heuristic will look for every revision in the software evolution path where *c* has been modified. The number of different revisions is the number of changes the element underwent – and, the higher that number, the higher the element's priority.

Inputs. In order to calculate the results for this heuristic, the only required input is the change sets that occurred through the systems' evolution. That is, the list of existing revisions and the code elements that were modified on each revision.

Motivation. Previous studies have investigated the relations between code anomalies and software change proneness. Khom *et al.* (2009) observed that the presence of code anomalies increases the number of changes that classes undergo. Changes to anomalous classes tend to trigger ripple effects through classes they are coupled to. Therefore, these changes might be dispersed through many classes, whereby increasing the likelihood of affecting architecturally relevant elements.

For instance, when analyzing previous systems looking for code anomaly patterns (MACIA *et al.*, 2012b), we realized that several anomalous code elements were related to either communication–related modules (like Façades and Proxies) or interfaces. We also observed that anomalies infecting those elements are perceived as more harmful than those found on private methods or classes that do not communicate with other modules (ARCOVERDE *et al.*, 2011). Moreover, through the system evolution, classes responsible for implementing APIs and Facades are more likely to change, as functionalities are added or modified.

Many types of code anomalies – such as God Class, Long Methods and Long Classes (FOWLER *et al.*, 1999) – are related to code elements that concentrate too many responsibilities, violating the *Single Responsibility Principle* (MARTIN, 2002). When that happens, the code is susceptible to change whenever any of the implemented responsibilities requires modifications. These responsibilities might be related to those realized by architecturally relevant elements, as the ones mentioned above. Therefore, intuitively, those code elements are more subject to changes through the evolution of the system implementation.

3.1.2. Error-proneness Heuristic

This heuristic is based on the idea that code elements that presented more errors through the evolution of a system might be considered high-priority.

Definition. Given a resolved bug *b*, this solution will look for code elements *c* that were modified in order to solve *b*. The higher the number of errors solved as a consequence of changes on *c*, the higher the position of *c* in the prioritization ranking.

Inputs. There are two necessary inputs for this heuristic: first, the set of bugs reported for the software under analysis; second, the list of code elements that were changed in order to solve each of the reported bugs.

Motivation. Recent studies have shown that software measures are strongly related to errors, and might be useful indicators for bug prediction (COUTO *et al.*, 2012). D'Ambros *et al.* (D'AMBROS *et al.*, 2010) studied the impact of code anomalies on software errors. They found that there is a correlation between the number of errors and occurrences of code anomalies. Moreover, there is empirical evidence that code anomalies and architecture problems are strongly related (MACIA *et al.*, 2012b). Thus, the occurrence of errors on anomalous elements might be related with architecture problems. For instance, complex elements are more likely to present a high number of errors (MACCABE, 1976). That can be an architecture problem when those complex elements are implementing modules interfaces. In this case, the errors can be propagated to several client modules.

Furthermore, Kim and Kim (2010) studied the role of refactoring on software evolution, investigating to what extent it was related to errors. The authors found that the number of bug fixes often increases after refactorings, whilst the time to fix them decreases. Moreover, their results show a relevant relation between incomplete or incorrect refactorings and bugs. Weißgerber and Diehl (2006) found that, following some revisions where refactoring took place, there was an increasing ratio of bugs reported. Those findings motivated us into analyzing to what extent bugs and code anomalies could be related.

3.1.3. Anomaly Density Heuristic

Another factor we take into consideration in the prioritization anomalies is the number of anomalies found per code element. For this heuristic, elements with more anomalies are considered high-priority targets for refactoring.

Definition. Given a code element *c*, we detect the number of code anomalies that *c* contains. The higher the number of anomalies found, the higher ***c*** will rank in the prioritization heuristic result.

Inputs. The only input needed for this heuristic is the set of detected code anomalies for the system under analysis.

Motivation. Our previous studies have shown that code anomalies and architecture problems are deeply related (MACIA *et al.*, 2012b). Frequently, a high number of anomalous elements concentrated in a single component indicates a deeper maintainability problem. Therefore, the classes internal to this component with a high number of anomalies should be prioritized. For example, multiple occurrences of Long Methods and Feature Envy (FOWLER *et al.*, 1999) in a single class are a recurring anomaly pattern that frequently indicates architecture problems. This heuristic would identify those elements as high-priority, as they present a high number of anomalies.

Furthermore, it is known that developers seem to care less about classes that present too many code anomalies (Broken Window Theory, MARTIN, 2008), when they need to modify them. Thus, anomalous classes tend to remain anomalous or get worse as the systems evolve. Their anomalous structure might also be the cause of increased complexity of their client classes, for instance, if the former ones are the sources of Feature Envy instances and method signatures are complex. Prioritizing classes with many anomalies should avoid this perpetuation and propagation of problems. This heuristic would also probably be worthy when classes have become brittle and hard to maintain due to the number of anomalies infecting them.

3.1.4. Architecture Role Heuristic

When architecture information is available, the architecture role a class plays on the overall architecture model influences its priority level. For example, on systems that follow the MVC pattern (BUSCHMANN *et al.*, 2007), a class might play three different roles – Model, View or Controller.

Definition. Given a code element *c,* this heuristic works by examining the architecture role *r* played by *c*. The relevance of *r* in the systems' architecture represents the rank of *c*. That is, if *r* is defined as a relevant architecture role, *c* will be ranked as high-priority.

Inputs. There are three essential inputs that the architecture role heuristic requires in order to calculate its results: first, the set of existing architecture roles in the software under analysis. Second, the traces between code elements and the architecture roles they play. Third, the level of relevance each architecture role should have.

Motivation. We saw in previous studies that high-level refactorings are commonly neglected, whereas refactoring of private members is often prioritized (ARCOVERDE *et al.*, 2011; MURPHY-HILL *et al.*, 2009). This happens because private members have no impact on external classes; thus, any changes on those methods will not be propagated to other code elements. Moreover, developers find high-level members harder to refactor, since they cannot predict the impact on client code (ARCOVERDE *et al.*, 2011). Therefore, the role played by a code element influences how and when developers apply refactorings.

3.2. Heuristics Scoring System

The aforementioned heuristics represent different ways to sort detected anomalies, according to their architecture relevance. In this context, the prioritization mechanism requires a scoring system for assigning concrete values to the code anomalies under analysis. Those scores represent the level of relevance those anomalies have according to each heuristic. For example, in the change-proneness heuristic, classes that suffered several changes will have higher scores than those that remained unchanged through the system's evolution.

All of the proposed heuristics are complementary; depending on the developer’s need, different heuristics may be more suitable for refactoring prioritization than others. Therefore, although they all work independently, we also provide a mechanism for combining the heuristics results, producing a single prioritization ranking. That combination may be weighted according to the user’s needs.

Section 3.2.1 details the scoring mechanism, in terms of how they are computed and represented. Section 3.2.2 describes a mechanism for combining heuristics results, which allows developers to apply all of the implemented heuristics at once, choosing different weights for each of them.

3.2.1. Computing Scores

The heuristics scores are numerical values associated with each code anomaly that represent how relevant that anomaly is with regard to the heuristic. That score is computed and assigned to the code anomalies by the heuristics themselves, through a scoring function. The set of code anomalies is sorted based on this numeric value in descending order.

The scoring functions inputs vary according to the heuristic that implements them. Nonetheless, as their purpose is to assign numeric values to code anomalies, they must all produce a standard well-formatted output. That output is a collection of pairs *<element, score>*. The element might be each anomaly within the project under analysis. That allows users to visualize their prioritization results from two different perspectives: i) from the most to the least harmful code anomalies and ii) from the most to the least degenerated classes, in terms of the anomalies that infect them.

It is important to notice that our current implementation is only able to distinguish priority levels between anomalies in different classes; all the code anomalies found in a given class will have the same score for all of the four prioritization heuristics proposed. However, this is a limitation of the current implementation – as the heuristics do not have such constraints conceptually. However, such limitation does not invalidate our results, as we shall see in

Section 4, since most of the architecture problems were related to groups of methods within the same modules.

We describe below how the scores are computed for each of the implemented prioritization heuristics.

Change-Proneness Heuristic. As we are only able to compute changes made to an entire file, for this scoring mechanism, all of the code anomalies within a single file receive the same score. However, we can still differentiate two different classes (as long as they belong to different files), ranking those that have changed the most as high-priority. The scores are computed according to the following algorithm:

```
changedFiles = repository.retrieveChanges(1,HEAD)
sortByDescendingOrder(changedFiles)

FOR EACH class IN changedFiles DO

    anomalies = class.getAnomalies()

    FOR EACH anomaly IN anomalies DO

           anomaly.setScore(class.getNumberOfChanges())

    END FOR
END FOR
```

In this heuristic, the score for each code anomaly is assigned from the number of changes that occurred in the infected class. In the example above, changedFiles represents an intermediary data structure that relates each file to the number of times they were changed in a given revision interval (e.g, from revision 1 to the head revision).

Error-Proneness Heuristic. For the *error-proneness heuristic*, scores are calculated similarly to the *change-proneness heuristic*; however, instead of assigning a number directly to the anomaly score, it is iteratively calculated from the number of times a given class appears on bug fixing revisions. The heuristic for identifying those revisions was explained in Section 3.1.2.

```
revisions = repository.retrieveRevisions(1,HEAD)
bugsFixingRevisions = filterBugFixingRevisions(revisions)

FOR EACH revision IN bugsFixingRevisions DO

    changedClasses = revision.getChangedClasses()

    FOR EACH class IN changedClasses DO

           FOR EACH anomaly IN class.getAnomalies() DO

                  result.incrementScoreFor(anomaly)
```

```
            END FOR
    END FOR
END FOR
```

First, we retrieve those revisions where bugs were fixed. We then iterate over all the classes changed on those revisions, and increment the score for the anomalies that infect those classes. Therefore, if a given class was related to several bug fixes, their code anomalies will have a high score.

Anomaly Density Heuristic. The following pseudocode summarizes the algorithm for computing this heuristic's results:

```
FOR EACH class IN project DO

    anomalies = class.getAnomalies()

    FOR EACH anomaly IN anomalies DO

            result.incrementScoreFor(anomaly)

    END FOR
END FOR
```

A variation to this approach only computes the anomalies that are classified as architecturally relevant. The pseudocode for that variation is shown below.

```
FOR EACH class IN project DO

    anomalies = class.getAnomalies()

    FOR EACH anomaly IN anomalies DO

            IF (anomaly.isRelevant()) THEN

                    result.incrementScoreFor(anomaly)

            END IF
    END FOR
END FOR
```

For determining whether an anomaly is relevant or not, we rely on the SCOOP detection mechanism, as explained on Section 1.3.2. That mechanism separates the detected code anomalies into those that are architecturally relevant (either by belonging to some code anomaly pattern or identified through SCOOP's detection strategies).

Architecture Role Heuristic. The architecture role heuristic computes its scores by simply analyzing the components infected by each identified code anomaly. For example, if the anomaly belongs to critical architecture components, its score is increased. The mechanics are described below:

```
FOR EACH class IN project DO
    IF (class.getArchitectureRole().isCritical()) THEN
        anomalies = class.getAnomalies()
        FOR EACH anomaly IN anomalies DO
             result.incrementScoreFor(anomaly)
           END FOR
    END IF
END FOR
```

Currently, SCOOP is able to analyze projects that follow the MVC architecture pattern, classifying as critical all the elements that belong to the Model component.

3.2.2. Combining Heuristics

Each of the proposed heuristics results in a ranking of code anomalies, based on specific scores. However, there might be cases where combining different heuristics gives us a better result, in terms of its correspondence to which anomalies should be prioritized. For example, anomalies belonging to classes that were never related to bug fixes will have the same 0-value score under the fault-proneness heuristic; therefore, the results of other heuristics are crucial for a correct prioritization output.

For combining different heuristics, they must all comply with the same scoring rules: in that sense, all of them will give the anomalies a numeric score, which will be multiplied by the heuristic weight. All of the available aforementioned heuristics follow these rules.

The configuration of weights for each heuristic is up to the user, as different projects might have different prioritization needs: for example, projects without an issue tracking system should not use the error-proneness heuristic, which is the equivalent of assigning a 0-value weight for it. Finally, the weighted sum of the combined heuristics score results in the final prioritization score, as shown in the following function:

$$\boxed{P(a) = (h1(a) * wh1 + h2(a) * wh2 + ... + hn(a) * whn) / wh1 + wh2 + ... + whn}$$

Where P(a) is the final prioritization score for a given anomaly, hx(a) is the score for a heuristic hx and wx is the weight assigned to each heuristic.

3.3. Use Case Scenarios

A common use case scenario for a code anomalies prioritization mechanism would assist developers in finding out which anomalies are causing more problems than others, giving them a prioritized view of the detected anomalies. They could choose a combination of different heuristics for identifying harmful anomalies, such as change-proneness and number of anomalies found per module. In an integrated development environment, such as Eclipse, the prioritization result would open a View with a list of all the identified anomalies and their scores.

Other use case scenario examples are:

1. The project has a huge number of bugs, and it is important to analyze which changes to the architecture are needed for diminishing the error-proneness of the system as a whole;
2. Few classes are responsible for most of the architecture problems, containing a huge number of anomalies and the development team must concentrate refactoring efforts in a limited set of classes per revision;
3. Co-occurrences anomaly patterns are found on classes that represent a key role in the architecture design. For example, the Overgeneralized Code Anomaly pattern may occur when a God Class has lots of outgoing relationships (or dependencies).
4. Another possible use case scenario happens when developers are dealing with an unknown code base, or assigned to a new project. The code anomalies prioritization mechanism could show them which are the major problems of that system, and where they should invest more time when learning how to evolve that system.

3.4. The SCOOP Tool

The previous sections introduced the prioritization heuristics concepts, including their scoring scheme. This section describes how those elements are put together into a prioritization engine that extends SCOOP (Section 1.4.2), and its implementation details. Section 3.4.1 describes the prioritization engine's main components, and how it was integrated into the SCOOP architecture. Our prioritization engine was implemented in Java, using the Plugin Development Environment (PDE).

3.4.1. Architecture

The implementation of our ranking mechanism is currently attached to SCOOP, as it receives the detected anomalies as an input for applying the prioritization heuristics. Our engine processes the list of anomalies identified by SCOOP, running each of the configured heuristics over them.

The illustration below summarizes the main architecture components added to SCOOP:

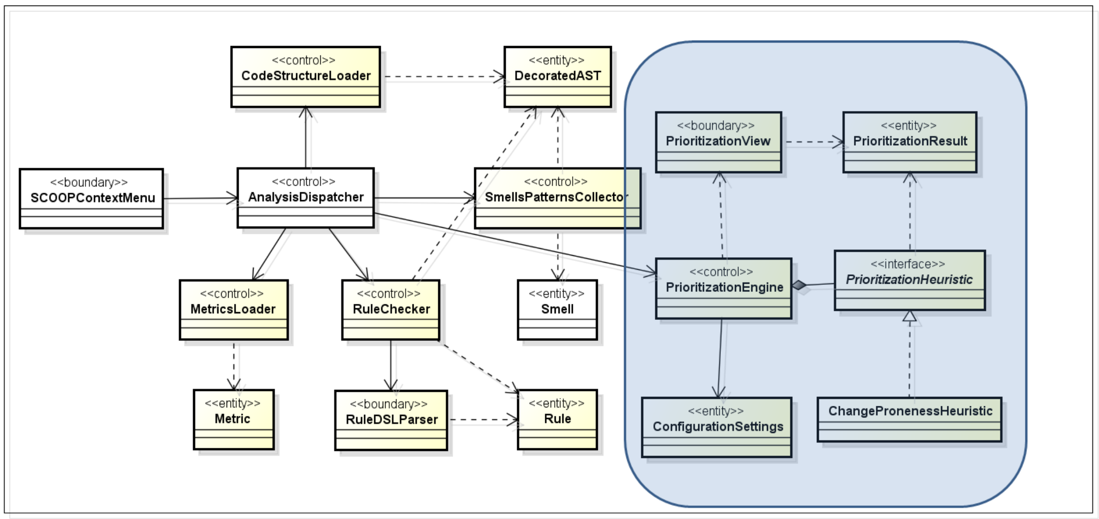
<<control>> CodeStructureLoader
<<entity>> DecoratedAST
<<boundary>> PrioritizationView
<<entity>> PrioritizationResult
<<boundary>> SCOOPContextMenu
<<control>> AnalysisDispatcher
<<control>> SmellsPatternsCollector
<<control>> PrioritizationEngine
<<interface>> PrioritizationHeuristic
<<control>> MetricsLoader
<<control>> RuleChecker
<<entity>> Smell
<<entity>> Metric
<<boundary>> RuleDSLParser
<<entity>> Rule
<<entity>> ConfigurationSettings
ChangePronenessHeuristic

Figure 3 - Prioritization components over SCOOP

The *PrioritizationView* implements the Eclipse View Part Extension – a graphical component that shows the prioritization results. These results are encapsulated in an entity called *PrioritizationResult*; every heuristic is implemented as a realization of the *PrioritizationHeuristic* interface. The *PrioritizationEngine* is a composition of heuristics, and controls the prioritization flow. It runs each heuristic and combines their results, consulting the *ConfigurationSettings* class. Finally, the entry point for the prioritization process is attached to the *AnalysisDispatcher* control, after the execution of SCOOP main workflow. We next describe the implementation details for each of the prioritization heuristics.

3.4.1.1. Change-proneness Heuristic

Currently, mainstream version control systems offer APIs for connecting to their databases and querying relevant information, like change sets and commit messages. There are, however, significant differences between the approaches adopted by centralized and distributed version control systems. Using Git (GIT, 2012), for instance, it is possible to download and analyze the entire repository, along with its change history. The analysis in this case is much simpler, as the changes can be locally retrieved. By using the Git Diff tool, it is possible to track file changes and even rename/move operations. This is a much harder task on centralized version control systems, such as Subversion (SUBVERSION, 2012), as they work with deltas between file changes, indexed by their names, instead of checksums.

Two mainstream source version control systems are currently supported by SCOOP's prioritization mechanism: Git and Subversion. None of them has a specific API for identifying which code elements where changed the most throughout the system evolution. Therefore, we implemented a component that retrieves this information from log files generated by both systems. For example, a log file for the SCOOP repository is shown in Listing 3.

Listing 3 – Example of Subversion log file

```
r93 | Roberta | 2012-01-13 02:09:00 -0200 (sex, 13 jan 2012)
Changed paths:
   M /br/pucrio/inf/les/genius/metrics/MetricsLoader.java
   M /br/pucrio/inf/les/genius/metrics/OtherMetricsCollector.java
   M /br/pucrio/inf/les/genius/metrics/TogetherCsvMetricsCollector.java
------------------------------------------------------------------------
r94 | Isela | 2012-01-22 21:49:17 -0200 (dom, 22 jan 2012)
Changed paths:
   M /br/pucrio/inf/les/genius/patterns/CodeAnomalyPatternCollector.java
------------------------------------------------------------------------
r95 | Isela | 2012-01-22 21:49:30 -0200 (dom, 22 jan 2012)
Changed paths:
   M /br/pucrio/inf/les/genius/archdesign/prolog/PrologConstants.java
   M /br/pucrio/inf/les/genius/gui/AnalysisDispatcher.java
   M /br/pucrio/inf/les/genius/gui/CodeAnomalyPatternPerspectiveFactory.java
   D /br/pucrio/inf/les/genius/gui/ConfigurationPage.java
   M /br/pucrio/inf/les/genius/metrics/CommonMetricsCollector.java
   M /br/pucrio/inf/les/genius/metrics/MetricsLoader.java
   M /br/pucrio/inf/les/genius/metrics/TogetherCsvMetricsCollector.java
   M /br/pucrio/inf/les/genius/metrics/classes/NumberOfUsedEnsemblesPerClass.java
   M /br/pucrio/inf/les/genius/metrics/classes/NumberOfUsedExternalClasses.java
   D /br/pucrio/inf/les/genius/patterns/CodeAnomalyPattern.java
   M /br/pucrio/inf/les/genius/patterns/CodeAnonalyPatternLoader.java
   M /br/pucrio/inf/les/genius/utils/PrologUtil.java
```

This log lists all the files changed for a given revision interval – in this example, for space constraints, we only show the results from revision 93 to revision 95. For identifying which files changed the most, we parse such log, counting the number of times each file appears in all of the listed revisions. The results are them sorted in descending order, from the files that appear the most to those that appear the least. For the example above, the *TogetherCsvMetricsCollector.java* file would be the one with the highest score for this heuristic, as it appears in the log file 9 times.

Our implementation of the change-proneness heuristic is limited to the analysis of changes that occurred to an entire file. That means that it is not possible to analyze change-proneness for distinct methods inside the same file. For example, it could be interesting to analyze whether certain methods are more subject to change than others, through more specific and detailed change-proneness reports. This could be achieved by parsing and analyzing change sets generated by the version control systems.

The implementation of the change-proneness heuristic follows the diagram illustrated in Figure 4. The *ChangePronenessHeuristic* class is responsible for executing the heuristic ranking mechanism. For doing so, it is connected to a version control repository, represented by the *Repository* interface.

The *Repository* interface is realized by two concrete implementations – *GitRepository* and *SubversionRepository*, responsible for retrieving information from Git and Subversion, respectively.

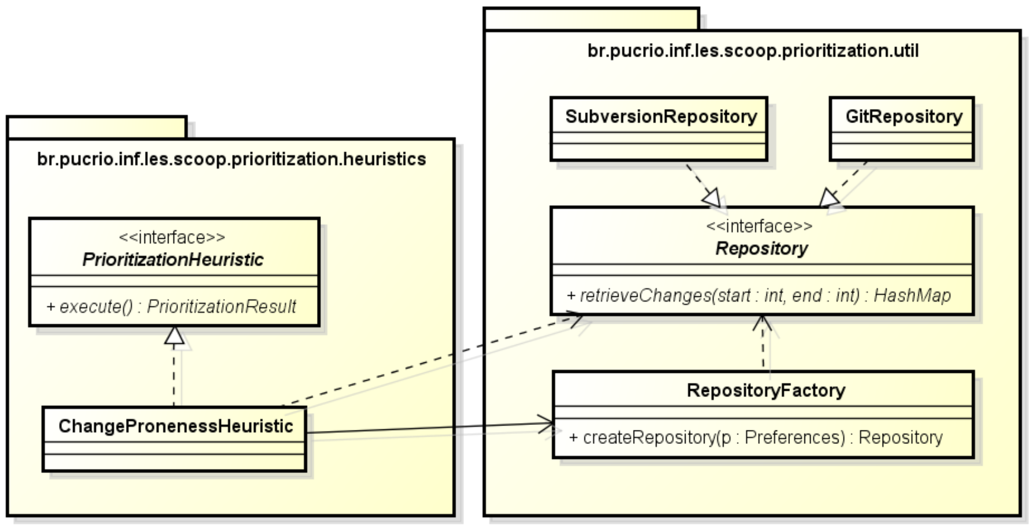
br.pucrio.inf.les.scoop.prioritization.util
SubversionRepository
GitRepository
<<interface>>
Repository
+ retrieveChanges(start : int, end : int) : HashMap
RepositoryFactory
+ createRepository(p : Preferences) : Repository
br.pucrio.inf.les.scoop.prioritization.heuristics
<<interface>>
PrioritizationHeuristic
+ execute() : PrioritizationResult
ChangePronenessHeuristic

Figure 4 - Change-proneness heuristic implementation

Besides connecting to the version control systems, each instance of *Repository* is also responsible for retrieving the changed files for a given range of revisions. That information is passed to the *ChangePronenessHeuristic* class, which calculates the scores for each element.

The *RepositoryFactory* implements the Factory design pattern (GAMMA *et al.*, 1995). This class is responsible for creating instances of the *Repository* interface, according to the user's preferences.

Although we have only implemented tool support for Subversion and Git, the change-proneness heuristic mechanism can be easily extended through the *Repository* interface.

3.4.1.2. Error-proneness Heuristic

In order to work properly, this heuristic depends on the correct use of issue tracking systems, such as Bugzilla (BUGZILLA, 2012) or Trac (TRAC, 2012). All of those systems provide APIs for querying their reports. Integration with source control systems is also needed, as most of current bug reports systems work with tags and unique identification keys for source control commit operations. For example, when committing a bug fix on Trac, the developer may specify the bug ID being solved. The source version control system triggers events that connect that commit operation to the bug report, closing it and establishing traceability between the modified files and the bug.

Mining bug systems is, however, far from trivial. Trac, for example, provides a querying API that allows programmatic integration with its tickets database. However, for using such API, a plug-in must be installed on the Trac server, a requirement that was not possible for the systems we analyzed. In order to overcome such challenges, we implemented a different approach for identifying code elements related to errors. SCOOP looks for comments on source control commit messages that possibly indicate the occurrence of bug fixes. This approach has been used on several studies (KIM *et al.*, 2011), and uses a simple text processing heuristic that looks for common terms such as "bug", "fixed" and "defect", as well as integration tags in issue tracking systems. For example, Trac tickets are commonly referred on commit messages by prefixing the message with a ticket number, enclosed by "[#" and "]", as shown in the example below.

Listing 4 – Commit messages integrated to Trac

```
r162 | renato.godinho | 2012-05-28 20:12:29 -0300 (seg, 28 mai 2012) | 1 line
Changed paths:
   M /trunk/src/PCA/Controllers/PainelController.cs
   M /trunk/src/PCA/Pages/Painel.html
   M /trunk/src/PCA/Util/WCFDispatcher.cs

[#51] Adding a PopUp for updating customers in the main screen
------------------------------------------------------------------------
r163 | renato.godinho | 2012-05-29 15:39:41 -0300 (ter, 29 mai 2012) | 1 line
Changed paths:
   M /trunk/src/PCA/Entities/Order.cs
   M /trunk/src/PCA/Controllers/PainelController.cs
   M /trunk/src/PCA/Scripts/Orders/NetworkUtil.js
   M /trunk/src/PCA/Scripts/Orders/PCA/Painel.js

[#48] [#49] Fixing orders update error
------------------------------------------------------------------------
```

In this example, the changes made on revision 163 were classified as a bug fix, from the comments left on the commit message (“Fixing orders update error”). Such assumption could be validated by inspecting the issues tagged in the message (#48 and #49) in the issue tracking system. In this context, the classes modified in such change would be candidates for the *error-proneness heuristic*.

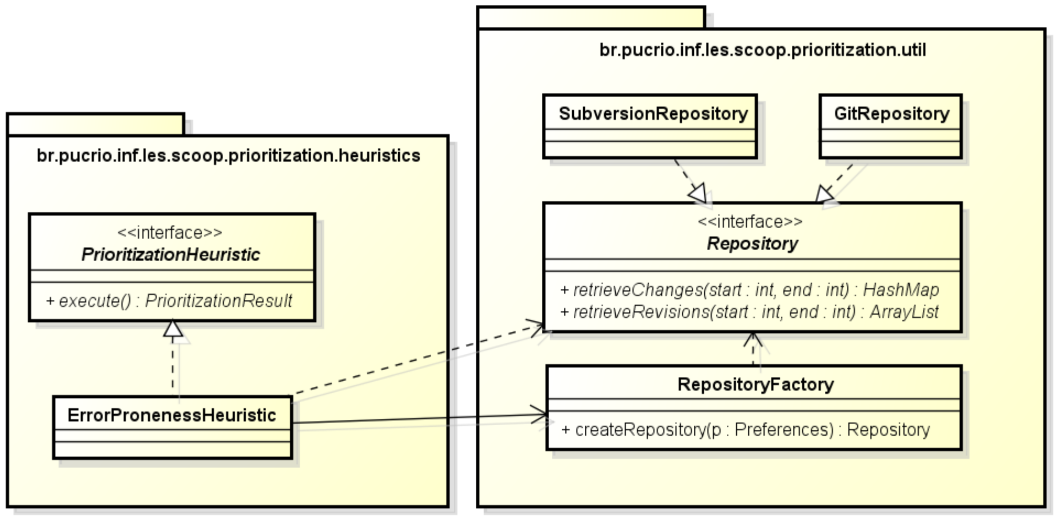
br.pucrio.inf.les.scoop.prioritization.util
SubversionRepository
GitRepository
<<interface>>
Repository
+ retrieveChanges(start : int, end : int) : HashMap
+ retrieveRevisions(start : int, end : int) : ArrayList
RepositoryFactory
+ createRepository(p : Preferences) : Repository
br.pucrio.inf.les.scoop.prioritization.heuristics
<<interface>>
PrioritizationHeuristic
+ execute() : PrioritizationResult
ErrorPronenessHeuristic

Figure 5 - Error-proneness heuristic implementation

The design for the error-proneness heuristic is shown in Figure 5. Similarly to the *change-proneness heuristic*, the implementation integrates with version control repositories for mining bug fixes and classes related to them. The *ErrorPronenessHeuristic* class is responsible for executing the heuristic ranking mechanism, calculating the scores for each anomaly under analysis. First, it retrieves the revisions for the project under analysis, as well as the files changed at each revision and the commit message associated with it. Them, the *ErrorPronenessHeuristic* class is responsible for filtering the revisions retrieved, looking for the following terms on commit messages: “bug”, “fixed”, “defect” and “error”.

3.4.1.3. Anomaly Density Heuristic

This heuristic was implemented as an extension point of SCOOP. Once the individual anomalies are detected, they can be ordered by the number of occurrences on each class or component.

Figure 6 depicts the main classes involved in the implementation of the *anomaly density heuristic*. As shown in Figure 6, this heuristic was implemented by the *AnomalyDensityHeuristic* class. That class is responsible for triggering the code anomalies detection mechanism implemented on SCOOP. The entry point for such mechanism is implemented by the *AnalysisDispatcher* class. Therefore, the *AnomalyDensityHeuristic* class first obtains the Java Project currently under analysis, passing its compilation units to the *AnalysisDispatcher*. The *CompilationUnit* class is part of the Eclipse Java Development Tool (JDT), a framework that provides an API for base manipulating Java abstract syntax trees (ASTs).

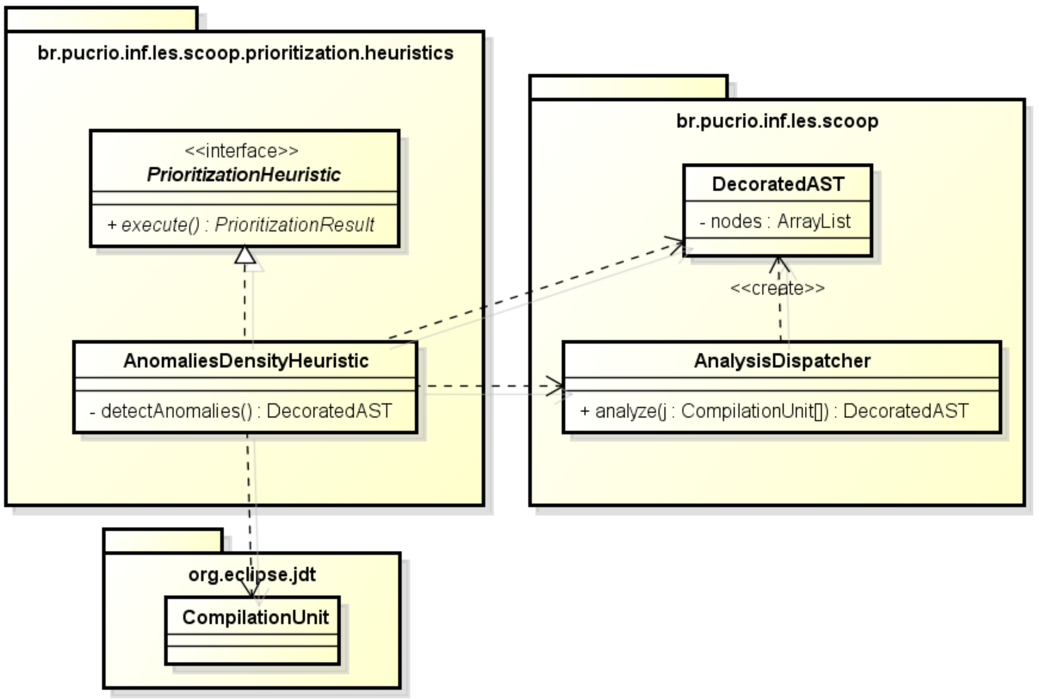
br.pucrio.inf.les.scoop.prioritization.heuristics
<<interface>>
PrioritizationHeuristic
+ execute() : PrioritizationResult
AnomaliesDensityHeuristic
- detectAnomalies() : DecoratedAST
br.pucrio.inf.les.scoop
DecoratedAST
- nodes : ArrayList
<<create>>
AnalysisDispatcher
+ analyze(j : CompilationUnit[]) : DecoratedAST
org.eclipse.jdt
CompilationUnit

Figure 6 - Anomaly density heuristic implementation

Once the code anomalies are detected, the *AnalysisDispatcher* returns an instance of the *DecoratedAST* class. That class implements the Decorator pattern (GAMMA *et al.*, 1995), adding the detected code anomalies to each node on the program AST.

Finally, the *DecoratedAST* is processed by the *AnomaliesDensityHeuristic* class. For each node, the class calculates the number of anomalies infecting it. The prioritization result will then be produced, containing the nodes with the highest numbers of anomalies found.

3.4.1.4. Architecture Role Heuristic

The *architecture role heuristic* mechanism ranks code anomalies based on the architecture role played by the elements they infect. In this context, it requires three inputs: (i) the existing architecture roles in the software project under analysis, (ii) a mapping between architecture roles and code elements and (iii) the relevance, or priority level, of each architecture role. The first two inputs could exploit the artifacts generated by architecture recovery tools (EICHBERG *et al.*, 2008; UNDERSTAND, 2012). For the heuristic implementation in SCOOP, users can use the Vespucci system (EICHBERG *et al.*, 2008) for providing that information. Alternatively, the architecture roles and elements related to them could be manually informed through SCOOP's API, as illustrated below.

```
model = new ArchitectureRolesModel()

model.addRole(“Business”, Priority.HIGH)
model.addRole(“Data”, Priority.LOW)
model.addRole(“ExceptionHandling”, Priority.LOW)

model.setRole(“Business”).toPackage(“br.pucrio.inf.scoop.business”)
model.setRole(“ExceptionHandling”).toClass(“util.BaseException”)

ArchitectureRoleHeuristic.setRolesModel(model)
```

As a default strategy, we implemented automatic ranking of architecture roles for projects that follow the Model View Controller architecture. Projects that do not follow this architecture pattern should have its architecture roles, and their relevancies, manually informed, through SCOOP's API.

For the MVC architecture, classes that implement the Model are considered as having higher priority than classes that implement Views and Controllers. For this implementation, we analyzed two different properties for categorizing classes as Views, Models or Controllers. First, we analyzed the hierarchy tree of each analyzed classes, looking for base implementations for those architecture roles. For example, the Spring MVC framework (SPRING, 2012) provides an abstract class named *AbstractController* that is the parent class for each controller implementation. The second property SCOOP observes is the name of the class under analysis; we looked for common suffixes, such as *View* or *Controller*, in order to identify the classes that played one of these roles. For example, a class named *UpdateClientView* would be categorized as playing the View role.

Figure 7 illustrates the main classes involved in the implementation of this heuristic. The *ArchitectureRolesModel* class is responsible for encapsulating the existing architecture roles, their relevance and the mappings between them and the code elements. As exemplified above, this class can also be manually instantiated, so that developers can inform their own architecture roles for the project under analysis. The *ArchitectureRoleHeuristic* implements the heuristic itself. It has a single instance of the *ArchitectureRolesModel* class, which is used for calculating the heuristic scores.

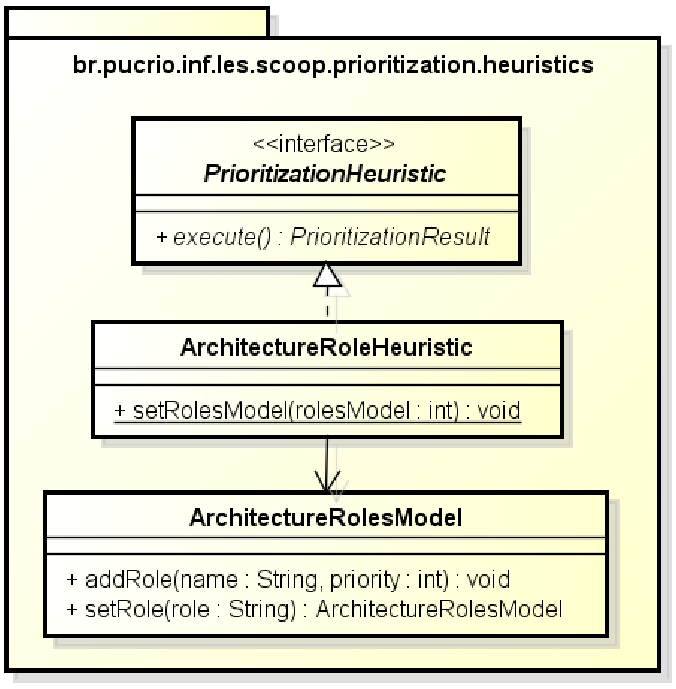
br.pucrio.inf.les.scoop.prioritization.heuristics
<<interface>>
PrioritizationHeuristic
+ execute() : PrioritizationResult
ArchitectureRoleHeuristic
+ setRolesModel(rolesModel : int) : void
ArchitectureRolesModel
+ addRole(name : String, priority : int) : void
+ setRole(role : String) : ArchitectureRolesModel

Figure 7 - Architecture role heuristic implementation

This chapter described the prioritization heuristics we propose for ranking code anomalies according to their architectural relevancies. In the next chapter, we describe the evaluation of those heuristics.