university of groningen

faculty of mathematics and natural sciences

Tool support for software metrics calculation of Java projects

Bachelor's Thesis Informatica

July 1, 2014

Student: J.D. van Leusen (2194767)

Primary supervisor: Prof P. Avgeriou, PhD Z. Li

Secondary supervisor: Prof M. Aiello

CONTENTS

1 INTRODUCTION

1.1 SOFTWARE METRICS & MEASUREMENT TOOLS

Software metrics are used to measure properties of software projects or software development. These properties range from the number of instruction paths in a function, to the number of lines of code in a project. Through the use of metrics these properties are quantified and used to measure various aspects, such as software complexity, correctness, efficiency and maintainability.

As an example of software metrics we can look at "McCabe's Cyclomatic Complexity" metric [4]. This metric is used to give an indication of branch complexity of a function by looking at the branching operators in that function, allowing it to give a rough indication of the overall complexity in a metric. Since functions with higher complexity are more prone to exhibiting bugs, this metric can be used to get an overview of complexity within a larger system by looking at individual methods in that system and calculating their complexity.

In order to apply these metrics in practice, tools are developed to calculate and to export resulting metrics based on static analysis of source code. The most common approaches are to parse a source file into an Abstract Syntax Tree (AST)[1] [5] and then to go through the structure of the AST to calculate the metrics. Alternatively simple tokenization of the source code is used for situations where linear parsing is enough to calculate the metrics.

Besides metrics that are applied to single source files, there are situations where metrics needs to be calculated over a collection of source files. These metrics are used to produce measurements of properties for an entire project or subset of a project by looking at the interaction between source files and comparing between source files. Metrics of this type require tools that take into account the global impact of a source file, tracking references and interactions between source files to produce results that take all gathered information into account.

Tools in general also support a method of use that makes them easy to integrate into the building process, allowing metrics to be tracked through development by calculating metrics for each new version of the source code and storing the results over time.

1.2 PROBLEM DEFINITION

While there are currently metric calculation tools for Java such as *Metrics*[2] or *JDepend*[3], they are very basic and tend to focus on a single set of metrics. The goal of this project is to design a new metric calculation tool for Java that implements calculators for the Chidamber and Kemerer [3] metrics and the Abdeen, Ducasse and Sahraoui [1] metrics. The modularity metrics defined by Abdeen, Ducasse and Sahraoui will need to be implemented based on the original paper due to the fact that there are no available tools that provide an implementation. The tool should also allow the user to easily extend it through the addition of new metrics.

The metrics defined by Chidamber and Kemerer measure the object-oriented design of a project. By measuring inheritance, relations between classes and class size, the complexity of the design at **class** level can be

[1] A tree representation of the abstract structure of source code, derived using the definition of the programming language syntax

[2] `http://metrics.sourceforge.net/`

[3] `http://clarkware.com/software/JDepend.html`

quickly determined. The modularity metrics measure the modularity of the packages contained within a project. In large projects packages tend to provide a service, hiding their implementation and giving the system a small set of interfaces to use. By measuring the implementation of these interfaces and the relations created between packages, the complexity of the larger system can be quantified.

The C&K metrics measure quality of detailed design, while the modularity metrics measure the system level software quality. This means the tool will need to be able to quantify software quality at different levels.

The target source code is delivered in the form of Java byte-code (.class files) or Java ARchive (JAR) files to avoid dependencies in the development environment, as well as allowing the tool to support calculation of metrics for applications without access to the original source code. The user should have the ability to use the tool through a Graphical User Interface (GUI) or be able to automatically run the tool through the command-line. The tool should avoid platform-dependent libraries to maintain cross-platform support.

The goal of this thesis and the project is to research the best way to create an extensible metric calculation tool for Java and then implement this tool together with the metrics mentioned earlier in this section.

1.3 STRUCTURE OF THIS THESIS

The remainder of this document begins with the chapter 2 'Background', which provides information about the libraries and projects used in the development of the final tool. The main goal of 'Background' is to give background information on these libraries and projects so their use within the project is clear.

The next chapter is chapter 3 'Concept'. It describes the general design of the tool and categorizes the metrics that have to be implemented. 'Concept' aims to provide the theoretical base and background for the tool's design.

Chapter 4 'Realization' describes the development of the tool, going over the decisions made and issues encountered during implementation of the tool. Its goal is to describe the transition from the theoretical base to the implementation and the way certain theories are applied.

Then comes chapter 5 'Evaluation', which looks into the performance and correctness of the results produced by the final version of the tool. It does this by going over the time it takes to inspect projects with varying sizes as well as comparing the results produced with the results from the reference implementation of the metrics. 'Evaluation' can be considered a review of the tool, checking how well the tool fits its requirements as well as check the implementation for mistakes.

Finally chapter 6 'Conclusion' is a review of the development process of the tool, discussing what the limitations of the implementation are and what could expanded on if someone were to continue work on the project.

2

BACKGROUND

For the purpose of development of the tool associated with this thesis, a number of libraries and projects are used. This chapter serves to provide a description and explanation of features for each of those libraries and projects.

2.1 APACHE COMMONS BCEL

The Apache Commons Byte Code Engineering Library (BCEL)[1] is a library that allows the runtime parsing and generation of Java byte-code. This means that the library has two major ways to use it: (1) Extracting data from existing class files, and (2) runtime modification or generation of classes, which is used for some high-performance libraries.

The use within the project focuses purely on the former. By using BCEL we can extract all data related to the class structure and its definition directly from the .class files. Invoking the parser creates a representation of the class, and this representation implements the **Visitor pattern**[2] which makes the process of deriving data from this representation fairly easy.

Classes in Java are resolved dynamically through the use of Class Loaders, so to retrieve all structural information BCEL has to mimic these Class Loaders. It does this through the concept of a 'repository', which acts as a pool from which it can retrieve the parsed classes and parse additional classes. To correctly use BCEL the repository needs to be well-defined or the library might not be able to find the byte-code it needs to fully parse a class.

2.2 GUAVA

The Guava project[3] is a set of libraries written and maintained by Google. They serve to extend and support the default Java API by providing utility classes and additional extensions of the default **collections** API.

Guava contains a couple of libraries that are of interest for this project, which are succinctly covered below.

2.2.1 *Basic Utilities*

Most of the utilities in Guava are fail-fast, if they are given incorrect or null input they will immediately fail instead of trying to give a correct answer. This property helps immensely during development since the use of undefined behaviour causes immediate failure, rather than unexpected results. This leads to less bugs in the source code during development.

Secondly Guava contains a set of classes that can be used to validate assumptions, this means that public methods can have enforced contracts, failing on invalid input just like their counterparts in Guava.

2.2.2 *Collections*

The Collections library consists of two parts, the first being a set of utility classes that support the existing Java collections API by providing generic

[1] `http://commons.apache.org/proper/commons-bcel/`

[2] The visitor pattern is a pattern where the data-structure is a tree and it has a 'visitor' interface that gets called for each node, which can then be automatically invoked by the data-structure

[3] `https://code.google.com/p/guava-libraries/`

constructors and operators, making the use of collections within the source code clearer.

The second part is a set of new collections and Immutable versions of all built-in collections. The Guava-supplied **Table** object which represents a map with 2 keys is a lot clearer in use than the equivalent map of maps. It better represents the type of data that it contains as well as being safer in use, since a Null-Pointer Exception (NPE) can happen very quickly with a map of maps.

The immutable collections are invaluable when working with concurrently accessed data-structures, ensuring thread-safe use of the collections as well as enforcing thread-safety in the code that operates on the collections.

Lastly the collections library adds functions that make some theoretic operations on collections a lot easier, an example of this would be the **Set**'s **Union** and **Intersection** operations, which are invaluable when implementing metrics involving a large amount of sets and set theory.

2.2.3 *Functional Idioms*

Java 8 adds support for lambdas and functional constructs to the Java API, through the use of Guava a lot of the functionality can already be used in this project. (Running in Java 6)

Since a lot of the collection operations within this project involve either transforming or filtering a collection, a functional approach makes a lot of sense and also results in cleaner code. The functional idioms **map**, **flatmap** and **filter** will be used a lot within the tool.

2.3 APACHE LOG4J 2

Apache Log4j 2[4] is a logging API. It differs from the default Java logging API by its performance and configurability.

Through its eXtensible Markup Language (XML) configuration all logging messages from the BCEL interface are filtered to only show important messages, not showing the debug output for this subset of classes. All logging is also put into a rotating log file, creating a unique file for each execution of the software and making it easier to find the logging associated with an execution of the tool. Lastly the framework is used to create a console within the tool where all important messages are shown. This console has to ignore exception stack-traces to be readable and through the configuration it can be selectively set to do so.

2.4 CKJM

Chidamber and Kemerer Java Metrics (CKJM)[5] is the reference implementation of the metrics defined by Chidamber and Kemerer [3]. It was created by Diomidis Spinellis for the purpose of a paper [6].

The implementation uses BCEL (see section 2.1) to parse Java byte-code and then calculates a set of metrics based on the paper mentioned above. It is designed to be used either as a Unix tool, giving the input through the standard input and outputting the results through the standard output or through invocation by the Java build-tool **Ant**.

While this makes the application's design quite simple, the fact that it is entirely driven through the standard input means it is not the easiest tool to use. This is especially true for bulk evaluation of large projects, since creating a list of classes to evaluate is completely left to the user.

[4] `http://logging.apache.org/log4j/2.x/`
[5] `http://www.spinellis.gr/sw/ckjm/`

The project does provide an example on how to implement each of the CK metrics, listed below.

- Weighted Methods per Class (WMC) - Normally a sum of the complexities of all methods in a class, the paper specifies that the constant 1 can be used instead of the complexity. This is the approach CKJM uses, so its implementation is simply an increasing counter for each method.
- Depth of Inheritance Tree (DIT) - CKJM defines this as the length of the list of all superclasses of the inspected class. Through the data that BCEL exposes this is easily obtained.
- Number of Children (NOC) - Calculated by having each class increment this value for its direct superclass.
- Coupling between Object Classes (CBO) - Calculated by creating a set of all classes that are used by a class, this includes the superclass, interfaces, return types, casts, fields, local variables and method arguments. Classes that are part of the Java API are filtered for clarity, since many classes use them implicitly.
- Response for a Class (RFC) - Is calculated by creating a set including all the signatures of methods that are defined and invokes within the class, the size of this set is the resulting RFC for the inspected class.
- Lack of Cohesion in Methods (LCOM) - Implemented by tracking the use of class fields within each method. Once it has gone through the entire class the fields used in all methods are compared pair-wise, incrementing LCOM if they share no fields used and decrementing LCOM if they share use in a field. If the final LCOM is negative it is set to 0.
- Afferent Couplings (CA) - Implemented as the reverse-mapping of CBO, for each relation found by CBO its inverse is registered with the class it targets.
- Number of Public Methods (NPM) - Implementation is the exact same as WMC, except it first tests if the method has the **public** modifier.

2.5 THE MODULARITY METRICS

The modularity metrics are defined in the paper by *Abdeen, Ducasse and Sahraoui* [1]. It defines a set of metrics that quantify the modularity of Java packages, considering each package a separate component that provides a clearly defined service to the larger system.

Each class defines two sets of relations with other classes. The *Uses* set of a class consists of all classes that are not a superclass of the inspected class, and are accessed through either method invocation or field access within the inspected class. The *Extends* set consists of just the direct superclass of the inspected class.

Once the *Uses* and *Extends* sets have been calculated for all member classes of a package, a union of the sets is created to represent the relations of the entire package. The paper defines **predicates** using these sets that represent the relationships between the package, classes and other packages.

Using the paper and the data defined above, a theoretical implementation can be defined for each of the modularity metrics.

- Index of Inter-Package Usage (IIPU) - Only defined for an entire system, not individual packages. It is calculated by calculating a set of all classes that are in the *uses* set of another class. This set is then copied and any classes that are only used by classes in the same package are

removed. The ratio between the sizes of these two sets is considered the *IIPU* of the system. This ratio describes the relative amount of cross-package class use, which should be as low as possible.

- Index of Inter-Package Extending (IIPE) - Same implementation as *IIPU*, but uses the *extends* sets instead of the *uses* sets.
- Index of Package Changing Impact (IPCI) - Given a package, the *IPCI* of that package is ratio between the number of packages that use that package and the total amount of packages in the system. If this value is high it means that any change to that package has impact on a large number of other packages in the system. The *IPCI* of the entire system is the mean *IPCI* of the packages contained in that system.
- Index of Inter-Package Usage Diversion (IIPUD) - The *IIPUD* of a package is calculated by observing the ratio between the number of packages and the number of classes in the *uses* set of that package. If this ratio is very high, it means that a package is using a lot of classes spread out among a large number of packages. This indicates a complex relationship between the package and the services that package requires from other packages. The *IIPUD* of the entire system is the mean *IIPUD* of the packages contained in that system.
- Index of Inter-Package Extending Diversion (IIPED) - Same implementation as *IIPUD*, but uses the *extends* sets instead of the *uses* sets.
- Index of Package Goal Focus (PF) - *PF* is calculated by observing how the package is used by other packages. For each of the packages that use classes in the inspected package, the amount of classes they use is compared to the amount of classes used by any package. If a package has a high package goal focus, it means that the same set of classes is consistently used by other packages. This means the package provides a focused service to the larger system. The *PF* of the entire system is the mean *PF* of the packages contained in that system.
- Index of Package Services Cohesion (IPSC) - *IPSC* is the alternative to *PF*, where a package provides multiple services. The *IPSC* of a package is calculated by observing what groups of classes are used by other packages and if those groups share classes. In other words, when comparing how a package is used by two other packages, measure the amount of classes shared by their use compared to the total amount of classes they use. If classes are shared between use-cases, it is implies the services are similar in purpose. This might mean that the service is not well-defined. The *IPSC* of the entire system is the mean *IPSC* of the packages contained in that system.

3 CONCEPT

This chapter serves to document the project requirements, the architecture design that the software will be built on and finally how the metrics mentioned in the requirements could be implemented.

3.1 REQUIREMENTS

- The project involves calculating a set of metrics for a given Java project.
- The input is provided as a set of JARs and Class files.
- The Byte Code Engineering Library (BCEL)[1] is used to extract information from the provided class files, providing information on the level of a **method**, **class**, **package** and **jar file**.
- Interaction with the software should be possible through a Graphical User Interface (GUI).
- The tool should be able to export the calculated metrics into a spreadsheet format like CSV.[2]
- If there are errors with the implementation of a metric or the provided input data then the tool should be able to handle these errors gracefully, providing accurate and explicit feedback about the cause of the error.
- If possible, the architecture of the project should be designed so it can be unit-tested. This means that the project should favour a modular design, allowing each module to be tested separately.
- The source code of the tool should use the common Java-style for code formatting to improve understandability and readability of the code.
- It will run under Java version 6 and the dependencies will be managed by Maven.[3]
- The project should provide implementations for the metrics defined by Chidamber and Kemerer [3], as well as the metrics defined by Abdeen, Ducasse and Sahraoui [1].
- It should be possible for the user to extend the tool by implementing their own metrics.
- The tool should allow for the execution to happen automatically based on command line input or execution arguments, to allow it to be used in scripted environments.

3.2 PROJECT ARCHITECTURE

The main architecture for the project is illustrated by Figure 1.

The 'Core' component acts as a messenger between the user-facing frontend, the data-providing BCEL interface and the individual metric calculators. It processes the input from the user, moves the data from the BCEL interface through the various metric calculators and finally sends the results back to the user.

[1] `https://commons.apache.org/proper/commons-bcel/`
[2] Comma-separated values
[3] `http://maven.apache.org/`

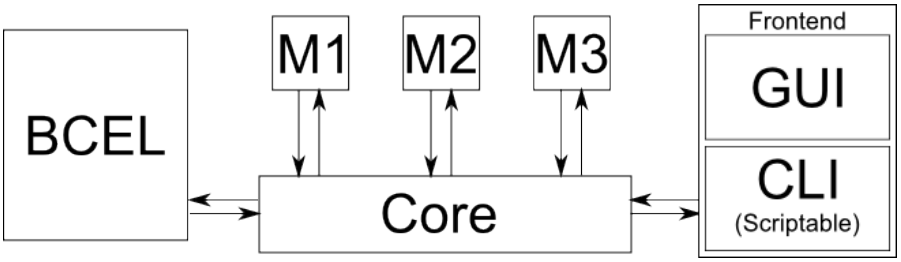


Figure 1: Abstract diagram of the architecture

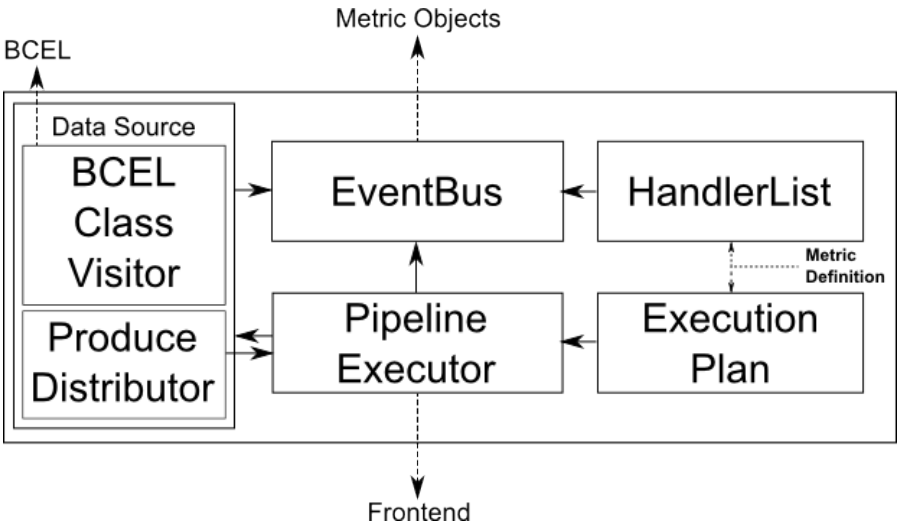


Figure 2: Abstract view of the components in the Core

The reason for this design is the fact that the project is highly modular, it is very easy to define components with dedicated tasks. This means that by designing the architecture around these tasks it is easy to both test each module individually and enforce the separation of concerns.

3.2.1 *Core, BCEL and In/Output*

The architecture is designed around the idea that the system works as a pipeline [2]. It processes each set of class files by passing it through the BCEL parser, passing the resulting data through the metric calculators and then returning the results to the front-end.

Within this design, the In/Output takes the form of 3 components that create the front-end. The GUI and CLI are designed to be interchangeable and serve as the interface with the user, letting the user select the target files and present the results. The third component is an exporting component which turns the results into CSV files.

The BCEL abstraction layer serves to implement the *Class Visitor* pattern to gather data from the class files, as well as gather auxiliary data like the JAR that the class originated from. Maintaining the source for a class is required because the Java language does not require a package to have just one source, so a unique mapping of package to container does not exist.

Tying the system together is the 'Core', shown in Figure 2. The core acts as an event bus, state registry and process manager for the calculation process. The task of this component is to ensure all the data it has to process gets passed on to the metric calculators correctly and to maintain the temporary state during calculation. The task of the process manager is also very important because there are certain metrics which have a finalization step which requires all input data to be processed it can be executed, which needs to be enforced for both thread-safety and result correctness.

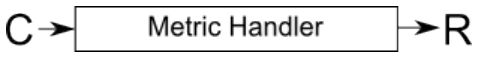


Figure 3: 'Isolated' metric (C = Structural Data, R = Metric Result)

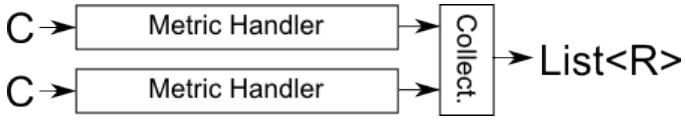


Figure 4: 'Shared' metric (C = Structural Data, R = Metric Result)

By abstracting the state of all the ongoing metric calculations into a central object, the entire system can be executed in parallel within this central point. This reduces the implementation complexity of the individual metric calculators since they only need to describe the process of deriving the metric from the data provided by the BCEL abstraction layer.

3.2.2 *Metric Calculator Design*

The most important part of the tool is the design of the metric calculators. The main purpose of these calculators is to describe the process of gathering the metrics from the provided data and produce a result once all the data has been processed.

In order to define a generic metric calculator, there needs to be a generic definition of a metric. When it comes to software metrics there are two major metric types with different calculation patterns.

The first type of metric is directly calculated from the data provided by the data source, data from other sources has no effect on the results of the metric. The second type of metric is calculated to take into account the impact of the data source. This means the result of the metric is dependent on the data from all data sources, not just the one associated with the result.

Defining the metric types in this way has several benefits. All metrics go through the same data gathering phase, deriving information from their data sources, but they differ in their finalization. The first type of metric can be finalized as soon as the data source has been fully consumed. The second metric type has to wait until all data sources have been consumed so the impact of all data sources can be taken into account.

While these basic types are enough to calculate any metric, metrics are usually calculated using a set of data defined by the creator of the metric. In order to avoid having to calculate this custom dataset for each metric, a supplementary metric type is defined that allows the user to pre-calculate a dataset they require for their metrics. These special 'producer' metrics can then be used as a shared data source for metrics that share that dataset.

In order to facilitate the creation of the various types of metrics, we define a set of base frameworks based on these generic metrics that define how the metric gets calculated within the data pipeline.

The first and simplest metric design is the 'Isolated' metric, which is illustrated by Figure 3. The main purpose of this design is metrics which can be directly calculated from a single data source without any outside information, like simply counting occurrences or usage of internal resources. Due to the fact that it does not require outside resources this type of metric can be calculated in parallel with minimal effort.

The second metric design is the 'Shared' metric, illustrated by Figure 4. This design is meant for metrics that have a global effect in their calculation, which requires the entire dataset to be ready before the calculation can be finalized. An example of metrics that require this last processing step would be a reverse mapping of method calls to get a set of users, since a class does not contain any information about its users and only contains what classes it uses. The design of this metric means all calculations that require just the

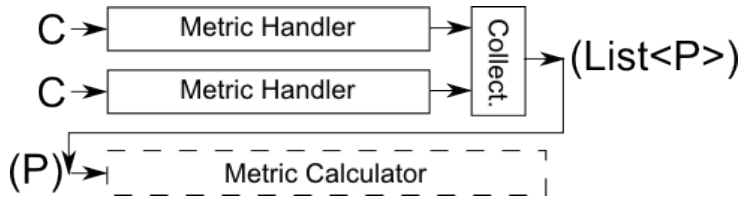


Figure 5: 'Producer' metric (*C* = Structural Data, *P* = Custom Data)

class data can be executed in parallel with a final synchronized finalization to calculate the results across all the classes.

The last metric design is a requirement to support groups of classes which represent packages or collections. This is the 'Producer' metric and is illustrated by Figure 5. The 'Producer' metrics are required because some of the higher-level metrics that are calculated for entire packages or collections require data that spans a set of classes, instead of individual classes. By moving the collection of metrics of that set to a separate calculator, the results can be reused for other metric calculations. This means that package-level metrics can be implemented by first implementing a collector for the package-specific data from the individual classes and then using the result as if it was data extracted from the class files themselves.

All the metric calculators are designed to consist of a data collection phase and a finalization phase, and by designing the metrics this way it is easier to design an efficient parallel processing model for the tool to follow when calculating the metrics. It also causes the metrics to be defined as a series of steps which calculate the result given a dataset and state, which encourages decomposition of the calculation into small data collection steps and a separate step which calculates the final result.

3.3 METRIC IMPLEMENTATIONS

To implement a metric a few factors need to be considered. First of all the required data has to be identified and registered to make sure it gets passed to the calculator at runtime.

After this is done handlers need to be written for each type of data, editing the state provided by the tool to work towards the result. Finally the finalizer needs to be written, transforming the state (or states for the 'Shared'/'Producer' metric) into a result that can either be communicated back to the frontend or used in other calculators if it is a 'Producer' metric.

CJKM

CKJM[4] is the reference implementation of the 6 metrics described in the paper by Chidamber and Kemerer [3], as well as two additional metrics. This means most of the implementation can be based on the reference implementation, possibly with some minor adjustments if the original paper describes a different method.

Weighted Methods per Class (WMC)

Metric type: **Isolated**.

In the original paper this is the sum of all the method complexities within the class, but CKJM simplified it so every method has a complexity of 1, effectively becoming the number of methods. The original definition could be a future extension of this metric but is not a part of the requirements.

The actual implementation of this metric is straight-forward, simply increment a counter for each method and report the final number as the result for this metric.

[4] `http://www.spinellis.gr/sw/ckjm/`

Depth of Inheritance Tree (DIT)

Metric type: **Isolated**.

This metric is simply a measurement of the depth of the inheritance tree from the class to the hierarchy root. This can be implemented by counting the number of direct superclasses of the class that is being inspected. For Java this is always 1 or greater, because all objects extend *Object*.

Number of Children (NOC)

Metric type: **Shared**.

This is the first metric which depends on data from other classes, this means the individual step for this metric is to store the superclass of the inspected class. Once the finalization step happens the superclass for each class needs to have their **NOC** count incremented by 1. This counter is the result for the associated class. If a class has no children it will be set to a **NOC** of 0.

Coupling between Object Classes (CBO)

Metric type: **Isolated**.

This metric can be calculated by maintaining a list of all classes references in the class. These references can be made by **method calls**, **field accesses**, **inheritance**, **arguments**, **local variables**, **return types** and **exceptions**. Simply maintain a list of all the types used in these references and the result is the length of this list.

Response for a Class (RFC)

Metric type: **Isolated**.

This metric can be calculated by maintaining a list of all unique method calls within the class and then returning the length of this list as a result. This is not the original definition but a simplification made by the author of CKJM because the original definition required a transitive closure of calls. The list of unique method calls can be obtained by looking for all method invocation instructions in the class and registering the target of those invocations.

Lack of Cohesion in Methods (LCOM)

Metric type: **Isolated**.

Calculation of this metric is a bit more involved, but the CKJM implementation provides a good definition. The implementation involves creating a set of used class fields for each method, which can be done by observing field access instructions and checking if the target class is the current class. Once all the methods have been iterated through, the result is calculated by first initializing the **LCOM** value to 0, then taking each unique pair of methods within the class and creating a union of the two sets of fields that they had created previously. If they share fields the **LCOM** is decreased, otherwise it is increased. This final value of **LCOM** is clamped to have a lower bound of 0 and is the result.

Afferent Couplings (CA)

Metric type: **Shared**.

This metric requires a reverse mapping of the **CBO** metric. So to calculate this metric we simply copy the **CBO** methods, but calculate the reverse mapping at the end. By counting all classes that reverse-map onto the current class, we can calculate the **CA** for that class.

Number of Public Methods (NPM)

Metric type: **Isolated**.

To calculate the NPM we simply observe each method in the class and increment the NPM value within the state if the method is marked as **public**. The value of NPM after all methods have been visited is the result.

Package Metrics

The metrics defined in the paper by Abdeen, Ducasse and Sahraoui [1] are applied on a collection level. However the specification of the metrics requires a new type of data that encapsulates the interface and references between a package and classes/packages outside of that package.

To make this information available to the metrics calculators defined in this section, we define a special Producer metric which acts as a data source for the metrics below.

The package metrics producer

Metric type: **Producer**.

To calculate the package and its references, we create a list of classes that are references in each class. The specification says that we need to maintain two lists for each class, one list containing the classes a class extends, while the other contains a list of classes it uses. These lists are stored and then reverse-mapped to prepare the data for the next step.

The classes are then bundled according to the package that they belong to and their references are split into internal and external references. The external references are then also reduced to a set of other packages.

Each package gets its own object encapsulating the classes it contains and the relations it has with other packages. All of the predicates defined by the paper are then turned into functions that return the relations specified by the paper. Through these predicate functions, the individual metrics can be implemented.

The reason a Producer is used instead of keeping all the metrics separated is that it would be a very large amount of repeat work to recalculate the above data for each metric, so the Producer is a construct to save work by preparing a set of data beforehand.

Finally, all of the metrics besides the $IIPU/IIPE$ metric are defined for both a package and a collection level metric, so both can be calculated and presented to the user. This is why all metrics in this section are **shared** metrics.

Index of Inter-Package Interaction (IIPU/IIPE)

Metric type: **Shared**.

To calculate this metric, we create a union of all the classes used by **packages** other than the package they are a part of. The length of this union is then divided by the length of the list containing all classes used by any other class. The resulting number is subtracted from 1 to create the **IIPU** value for the given collection of packages.

The exact same process is applied for the 'extends' lists, to calculate the **IIPE** value.

Index of Package Changing Impact (IPCI)

Metric type: **Shared**.

This metric is calculated by first calculating how many packages use a certain package. This is done by counting the number of packages that use a certain package and dividing this number by the total number of packages

minus one. The mean of all the values that result from that calculation is the **IPCI** value for the collection.

Index of Package Communication Diversion (IIPUD/IIPED)

Metric type: **Shared**.

Like the previous metric, this metric is calculated by taking the mean of a value calculated for each class in the set.

That value is defined as $\frac{1}{UsesP} \cdot (1 - \frac{UsesP-1}{UsesC})$ where $UsesP$ is the number of packages the selected package uses and $UsesC$ is the number of classes the selected package uses. If $UsesC$ is zero the value defaults to 1.

The mean of all these values is the **IIPUD** for this collection, the same process can be repeated using the 'extends' lists instead of the 'uses' lists to calculate the **IIPED**.

Index of Package Goal Focus (PF)

Metric type: **Shared**.

To calculate this metric we once again calculate the mean over a set of values that are calculated for each class.

For this metric, we need to calculate, from the set of classes that are used by other packages, how much of these classes are used by the other packages on an individual basis. The resulting percentages are once again used to calculate a mean which denotes the value for the selected package.

To calculate the share of classes used between packages, we take the subset of classes that are used by a specified package and divide the size of that set by the number of classes that are used by other packages globally.

The final value is the **PF** value for that collection.

Index of Package Services Cohesion (IPSC)

Metric type: **Shared**.

To calculate this metric, we need to define what is calculated. The basic calculation is that given the set of classes in p used in q, how many of those classes are used in a different set given a different pair p and k, $p \neq q \wedge q \neq k$. This number is calculated for each p and q, k pair, then the mean is calculated to produce the Service Cohesion for that package. The mean of all package produces the $IPSC$ for the collection of packages.

4 REALIZATION

This chapter serves to turn the theory described in chapter 3 into a working tool. It will cover the decisions made, reasons why those decisions were made and serve to document some of the issues encountered during development.

4.1 DESIGN TO IMPLEMENTATION

Implementing the theoretical design from chapter 3 can be seen as the construction of five major components:

- The interface with BCEL and the associated management of dependencies through a Repository.
- Turning the isolated, shared and producer metrics into Java objects that can exhibit all the properties discussed in the theoretical chapter.
- Given a set of metrics, figure out whether it can be evaluated and in which order the metrics would need to be evaluated.
- Implement the required metrics within the new framework.
- Handling exceptions during registration and calculation.

The implementation of these five components will be discussed in the following sections.

4.1.1 *Interfacing with BCEL*

As specified in the section discussing BCEL(section 2.1), the data objects produced by BCEL implement the **Visitor pattern**. The fact that they implement this pattern means it can be used to gather all the information required by overriding the methods that process the required data.

Given that getting the data required is not an issue, the focus shifts to the delivery of the data to the core and the management of dependencies of the data.

Because we have a piece of data and an unknown number of possibly interested metrics, the most obvious method of delivery of the data to the metric processors is through an **EventBus**[1]. These EventBusses are created by the system during runtime and provided to the BCEL visitor objects for the purpose of publishing information about that object. Registration of interest by the metrics can be done by the system before execution even begins.

Managing the dependencies of the data is done through a **Repository** object in BCEL. This object keeps track of already parsed classes and performs lookup of classes if their definition is required. One particular type of repository is of use within the tool. The **ClassLoaderRepository** which retrieves its data through Java Class Loaders.

By using this particular repository we can create a proxy-Class Loader that tracks which source the class came from, allowing the tool to see where a class was loaded from. A feature not normally present in a Class Loader. The decision to use this type of repository also allows for the use of the built-in **URLClassLoader** which can load classes from class files located in directories or a Java ARchive (JAR), allowing the user to specify an arbitrary number of sources of class files as input for the tool.

[1] An EventBus is a data structure that takes a piece of data and publishes it to a set of interested objects, these objects show their interest by registering it with the EventBus.

The metric types described in subsection 3.2.2 can be directly mapped to a set of abstract base classes. These classes can then require the user to provide an implementation of the final step of their process. (Either **getResult(s)** or **getProduce** depending on the metric type)

The implementation then shifts to creating a handler for the data published by the BCEL class visitor, as described in subsection 4.1.1. In Java data handlers can be implemented in two ways, either by using an interface or Java method annotations.

The first method, an interface containing all the data published by the system, allows the user to override the method if the user is interested in using that data for the metric calculation. Due to the fact that the **producer** metrics exist this type of design will not suffice since the user can define arbitrary new types of data, which would not be supported by this type of handler definition.

This means the implementation will use the second type of handler definition, Java method annotations. By creating an annotation that marks a method as a data handler, the system can create a list of interested metrics for the EventBus through Java reflection[2]. It also means we have an easy way to register the use of an producer metric through a second annotation that represents the intent to use the produce from that producer for the annotated method.

Listing 4.1: Metric definition with annotations.

```
1  public class SomeMetric extends IsolatedMetric {
2
3  @Subscribe
4  public void dataHandler(?, RequiredData data) {
5      ...
6  }
7   @Subscribe
8  @UsingProducer(ProducerMetric.class)
9  public void producerHandler(?, Produce data) {
10     ...
11 }
12
13 @Override
14 public MetricResult getResult(?) {
15     ...
16 }
17 }
```

As shown in the example metric above, one part of the metric definition is missing. This missing part is the way the system handled the temporary state of the metric during calculation.

One way of handling this would be to let the state be maintained within the metric calculator objects. While this would work for the Isolated Metrics this presents a problem for the Shared and Producer metrics, which require the state of all calculated targets[3] to create the final results.

This problem means it is a better idea to move the temporary state out of the metrics and into the system itself. This also means that the metrics objects purely represent the implementation of metric calculation and not maintain an internal state at all. By moving the state out of the metrics we can also consider use of the calculators thread safe, since all state is provided, isolated and managed by the system. See Listing 4.2 for an example of the final metric calculation definition design, the *invalidMembers* argument is used to tell the metric that failure has occurred while gathering data from that number of targets.

[2] Reflection is a system to examine and modify the behaviour of the program itself during runtime

[3] Targets in this tool refer to a set of data that belongs together, this can be a class, package or collection represented as a set of data.

Listing 4.2: Finalized metric definition, showing SharedMetric

```
1  public class SomeMetric extends SharedMetric {
2
3  @Subscribe
4  public void dataHandler(MetricState state, RequiredData data) {
5      ...
6  }
7  @Subscribe
8  @UsingProducer(ProducerMetric.class)
9  public void producerHandler(MetricState state, Produce data) {
10     ...
11 }
12
13 @Override
14 public List<MetricResult> getResults(Map<String, MetricState> states,
       int invalidMembers) {
15     ...
16 }
17 }
```

4.1.3 *Pipeline Design*

Now that an implementation for the metrics has been found, the issue becomes the evaluation of those metrics. It is a two part problem with the first part being the planning phase and the second part being the execution phase.

The goal of the planning phase is to take all the metrics and figure out in what order they need to be executed to ensure all the data has been delivered when they are finalized. This last part is important because of the addition of Producer metrics, which make their produce available only after they have been finalized.

After the planning phase, the execution phase needs to be defined. This phase takes the execution plan created in the planning phase and executes it, maintaining state and moving data between the data handlers and the finalization of the metrics.

The execution plan

The fact that metrics can use data produced by **producer** metrics means that dependencies between metrics exist and have to be taken into account. To ensure that the metric calculation can be finished the tool must assert there are no cyclic dependencies between metrics.

Since metrics can be defined for three different scopes, **Class**, **Package** and **Collection**, there also needs to be a one way transition between scopes due to the fact that data for a scope has no context within other scopes. Data can transition between scopes through **ProducerMetric**s.

All these constraints means the execution plan needs to take the form of a directed acyclic graph to ensure the system always moves forward and to ensure it always finishes. The implementation of this data structure within the tool is called **Pipeline Frames**, named that way because a single frame describes a frame of execution within the pipeline with a set of data to be delivered and a set of metrics that will be finished in that frame.

A simplified diagram of the **pipeline frames** system can be found at Figure 6. This diagram shows how data and execution flow through the defined frames.

To begin constructing the **pipeline frames**, the system creates a starting frame for each scope and registers all the data produced by the BCEL class visitor with the **class** frame. This is done because the information produced by the class visitor only has context within the class frame, losing context if that data were used in a **package** or **collection** frame directly. This base set of data can be extended if the user wants to add an extended class visitor.

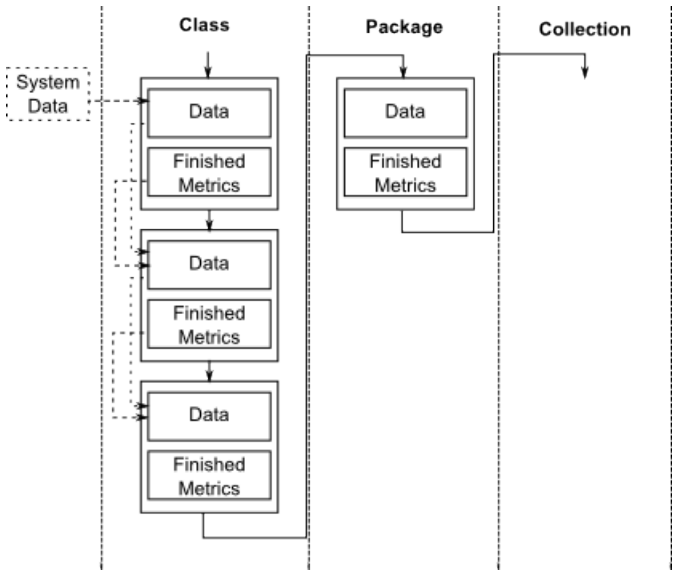


Figure 6: Diagram of the 'Pipeline Frame' execution plan

Every time a metric is registered the system first goes through all the methods of the metric, validating any method marked with the **@Subscribe** annotation. If the system encounters an **@UsingProducer** annotation, it checks if the referenced producer is not loaded already. If the producer does not exist it creates a new instance of the producer and registers it like it would for a normal metric, but in addition to registering the metric it also adds the produce produced by the producer into the set of available data of the frame **after**[4] the frame where the producer is finished executing, creating a new frame if it does not exist.

Because frames are executed sequentially, all data from a previous frame in the same scope is available in the current frame, creating the relation $previousFrame \subseteq currentFrame$. This is important because it allows the system to look for the first frame in which all of the data required by a metric is available, registering the metric for finalization within that same frame. This means that if a metric depends on a producer, it will be scheduled to finish in the frame after the producer since the data of the producer will be available in that frame, in addition to any of the base data required by the metric.

In essence this creates an execution pipeline, a sequenced set of data delivery and data creation which results in the calculation of the defined metrics.

This way of determining metric finalization means that situations where a metric requires data from different scopes or requires data that is not defined will not be able to find a suitable frame to be finalized in, providing a way to catch faulty definition during registration and enforcing the directed acyclic dependency requirement.

The actual execution

Given the execution plan defined in the last section, this section aims to define an execution system that executes the plan in the most efficient way possible.

At the beginning of execution, the system is supplied with an execution plan, a **repository** to load class data from, a BCEL class visitor factory[5] and a list of classes to inspect.

[4] If the producer produces for another scope, it will instead use the first frame of that scope
[5] The factory pattern defines an object that produces new instances the product object on request

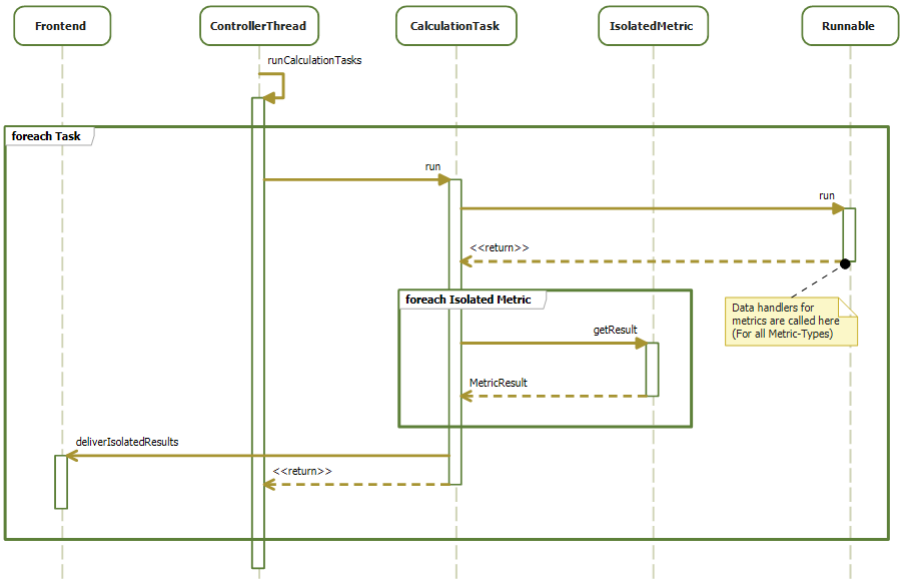


Figure 7: Sequence diagram of the Calculation step

Each frame has a list of **Runnable**[6] tasks that deliver a set of data to the metric calculators. For the first frame these tasks are Class Visitors for each of the input classes and for subsequent frames these are dispatchers for the Produce created by Producer metrics in the last frame.

This means the first thing to do during execution is to create the initial list of tasks by using the Repository to load the input classes and then creating a **class visitor** task using the provided factory. This initial set of tasks is used to begin the frame-loop execution, passing data to the metric calculators and creating new data and results through the defined metrics. The frame-loop starts at the first frame defined for the **class** scope and then moves through the frames defined for that scope, moving to the next scope once there are no more frames for the current scope.

High performance can be achieved by using the multi-core architecture of modern computers and doing as much work in parallel as possible. This comes into play in the first step of the frame-loop, where each unique target can be evaluated in parallel because the data delivered will always only concern the target it is associated with.

Once the data has been delivered by running the task, any **isolated** metrics defined for the current frame can be evaluated and directly sent back to the user. This can be done because the theory behind the specific metric, which specifies that isolated metrics do not require data from any other source. So if an isolated metric is marked for completion in the current frame, it can be completed immediately after the task has finished.

A sequence diagram representing the parallel calculation step of the frame-loop can be found at Figure 7.

The next step in the execution is to calculate the shared and producer metrics, but to do this the system first needs to extract all the state information from the individual targets so they can be delivered to the calculator as a single collection.

All state is maintained within the EventBus objects defined in subsection 4.1.2. The implementation contains a method that destructively extracts the state objects for a given set of metrics, removing them from the EventBus object since they will no longer be required after metric finalization. By doing this for all the EventBusses associated with the targets in the current scope a table of states keyed to a pair (Target,Metric) is created, which can be transposed to get a table keyed to (Metric,Target).

[6] Runnable is an interface in the Java API, often used to represent an task that can be executed asynchronously.

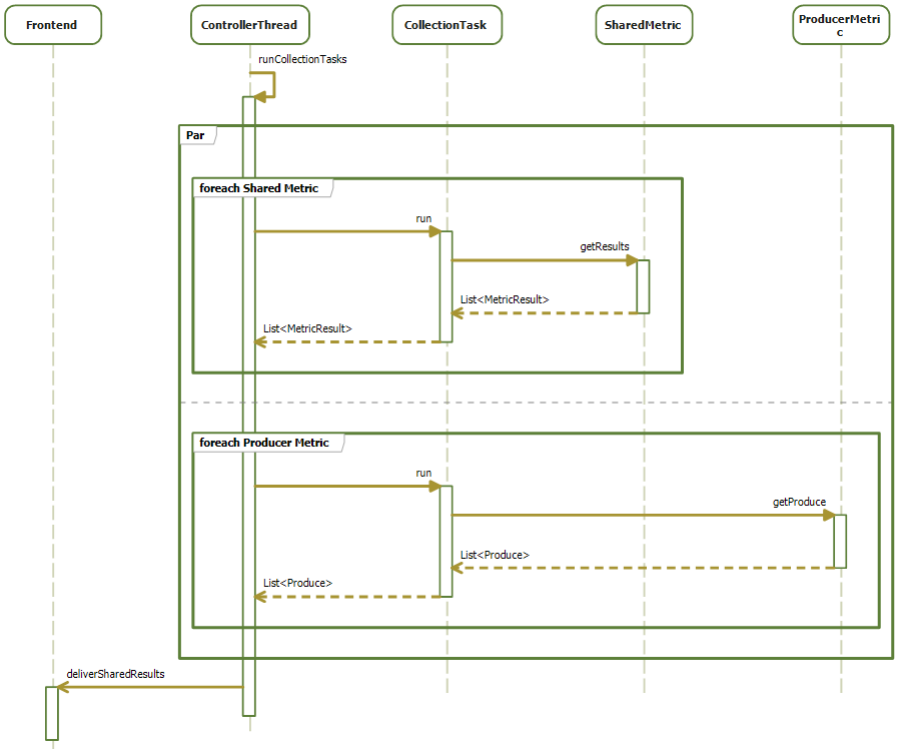


Figure 8: Sequence diagram of the Collection step

This method of gathering up all the state objects is used to gather the state for all the **shared** and **producer** metrics that are finished within the current frame. Then another parallel execution is performed by evaluating all the shared and producer metrics in parallel given their own set of data. Once all results and produce have been gathered, the results are sent back to the user and the produce is returned to the frame-loop for the next step of processing.

A sequence diagram representing the parallel collection step of the frame-loop can be found at Figure 8.

All the metrics for the current frame have been calculated and sent back to the user, but the produce from the **producer** metrics still needs to be handled. What happens to the produce depends on the scope it is meant to be delivered in. If the scope is the same as the scope of the current frame, it is queued for delivery in the next frame. If the scope is not the same then the produce is stored until execution of the produce scope begins.

Nearing the end of the frame-loop, the current frame is moved to the next one in the pipeline. If there are no more frames for the current scope, it switches to the next scope and the first frame of that scope, as well as wiping all state since it depends on the scope it is evaluated in and loses value after the scope has been switched. If the scope has been switched it will also queue up all the produce stored for that scope previously.

The final part of the frame-loop involves turning the produce that is to be delivered into tasks for the next frame. It does this by grouping all produce by target, then creating a **Dispatcher** task for each target and the produce that needs to be delivered to that target.

Once the last frame of the last scope has been executed, the frame-loop will terminate, the resources will be cleaned up and the user will be notified of that the calculation has completed. A sequence diagram representing the entire frame-loop can be found at Figure 9.

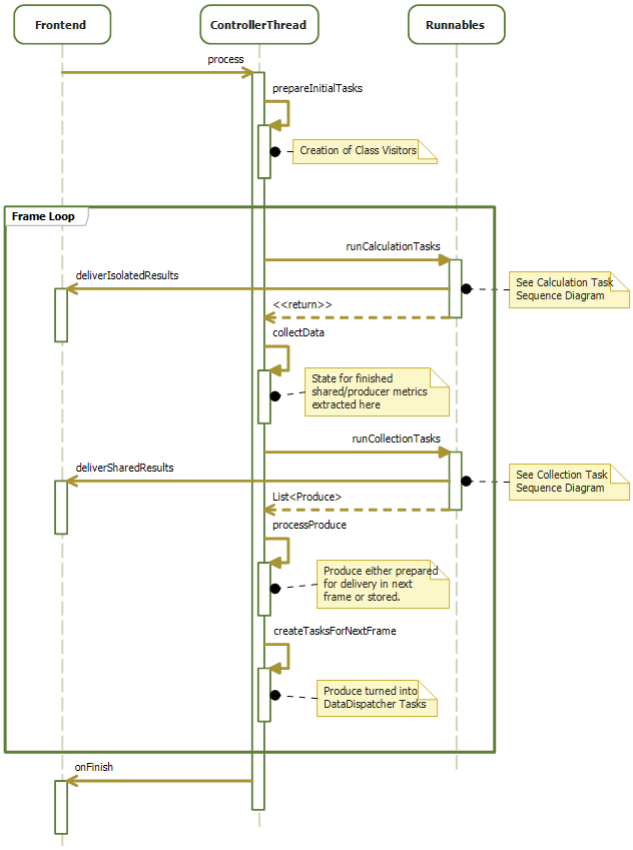


Figure 9: Sequence diagram of the Frame-Loop

4.1.4 *Defining the metrics*

Before implementing the CKJM and modularity metrics, this section covers the process of implementing a generic metric in **JSM**. The process is also shown in Figure 10 as a flowchart diagram.

The first thing to do when implementing a metric for JSM is to identify the basic data types required to calculate the metrics. This includes any data contained in the class byte-code and additional data like the container a class belongs to. If the calculation requires data that is not exposed by the default class visitor, that data can be exposed by overriding the default class visitor with an extended implementation that exposes the data. To use the custom class visitor, the class visitor factory used in the **PipelineExecutor** should be replaced before beginning execution.

If the metric is part of a collection of metrics and that collection has a shared set of data that is used in the metric definition, it is worth defining a **ProducerMetric** that pre-calculates the shared data. This producer gathers the data provided by the class visitor and publishes custom data for use in other metrics. By defining a producer metric, duplicate calculations are avoided by having all metrics share the pre-calculated data.

The next step is determining whether the metric follows the **isolated** metric archetype or the **shared** metric archetype. The distinction is made by the source of the data that the metrics are calculated from. If the metric can be calculated using just the data associated with the inspected target, that metric follows the **isolated** metric archetype and should extend the **IsolatedMetric** class from JSM. If the metric requires data from other targets to be calculated, an example being the use of a class by other classes, that metric follows the **shared** metric archetype and should extend the **SharedMetric** class from JSM.

Finally the metric needs to define its data handlers and finalization. The finalization methods are defined as abstract methods in the base classes,

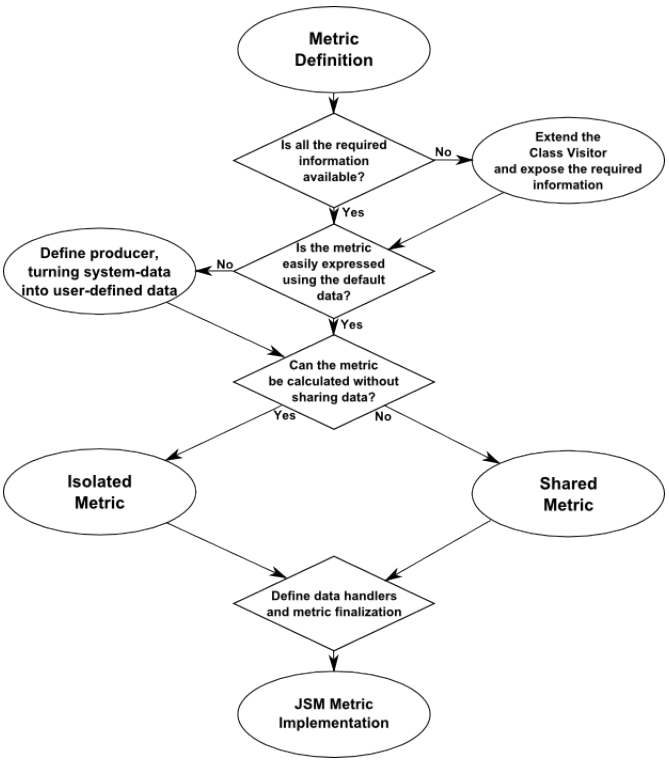


Figure 10: Flowchart diagram showing the process of implementing a metric in JSM

so they can be implemented by following the contracts specified in their documentation and signature. The user will receive the calculated state and should return a result or a list of results, depending on the base class that was used.

The metric calculators should have no state of their own, the calculation system maintains a set of state objects that the calculator can query and store information in. Maintaining state outside of these objects is not thread safe, since multiple targets could be processing and calling the methods of the calculator at once. The method definitions should follow the structure given in subsection 4.1.2. If the metric uses data created by **producer** metrics, it is recommended that the user annotate any handler of that data with the **@UsingProducer** annotation. This annotation tells the system to make sure that the **producer** metric is loaded and checks that the data the user is expecting matches the data produced by the **producer** metric.

4.1.5 *Implementing the metrics*

Now that the metric interface, the execution plan and the execution engine have been defined, the two metric packages specified by the requirements can be implemented.

Chidamber and Kemerer Java Metrics (CKJM)

Based on the CKJM implementation a set of base data types are exported by the default BCEL class visitor. This data is enough to calculate the metrics required by this project as well as any metric that requires information about relations between classes.

Most of the CKJM implementations with the exception of **CA** and **NOC** can be implemented almost directly as **isolated** metrics. The two metrics mentioned need to be implemented as **shared** metrics because they are affected by other classes. This means they will need to be rewritten to store

the required shared data until the finalization, instead of modifying the state of other classes while they are being evaluated.

In the case of **NOC** this is quite easy, each class stores the name of its superclass in its state. During finalization of the metric an integer for each of the targets (representing all the input classes) is initialized to 0. By then going through all the states and incrementing this value if the state indicates the class is a superclass the **NOC** for all the inspected classes can be calculated. This finalization step can be seen at Listing 4.3.

Listing 4.3: nl.rug.jbi.jsm.metrics.ckjm.NOC

```
44 final Map<String, Integer> nocMap = Maps.newHashMap();
45
46 for (final String className : states.keySet()) {
47 nocMap.put(className, 0);
48 }
49
50 for (final MetricState ms : states.values()) {
51 final String superclass = ms.getValue("superclass");
52 final Integer noc = nocMap.get(superclass);
53 //If its not a class we're inspecting, ignore it.
54 if (noc == null) continue;
55 nocMap.put(superclass, noc + 1);
56 }
```

The second CKJM metric that has to be rewritten as a **shared** metric is **CA**. While all of the listeners are the same as **CBO**, the difference is that unlike the CKJM version, the reverse reference is not created when a reference is found.

Instead, the metric builds up the same reference set as **CBO** and then performs a mass reverse-mapping during the finalization, creating a list of reverse references for each class which can then be used to determine the **CA** value. The code to implement this reverse-mapping can be seen below in Listing 4.4.

Listing 4.4: nl.rug.jbi.jsm.metrics.ckjm.CA

```
142 final Map<String, List<String>> reverseMap = Maps.newHashMap();
143 for (final String className : states.keySet()) {
144 reverseMap.put(className, Lists.<String>newLinkedList());
145 }
146
147 for (final Map.Entry<String, MetricState> entry : states.entrySet())
    {
148 final Set<String> coupledClasses = entry.getValue().getValueOrCreate(
    "coupledClasses", EMPTY_SET);
149 for (final String coupledClass : coupledClasses) {
150     final List<String> reversedList = reverseMap.get(coupledClass);
151     if (reversedList != null) {
152         //Can be null if coupled class is not in inspection scope.
153         //Guaranteed to be unique because states.keyset() is a Set
154         reversedList.add(entry.getKey());
155     }
156 }
157 }
```

Abdeen, Ducasse and Sahraoui's Modularity Metrics

Besides implementing CKJM, the modularity metrics defined by Abdeen, Ducasse and Sahraoui [1] have to be implemented. The metrics did not have a reference implementation, unlike the Chidamber and Kemerer metrics, which meant it had to be implemented straight from the paper.

The nature of the metrics meant that the use of a **producer** metric was heavily encouraged, due to the fact that all metrics are calculated using a set of relations between packages. By calculating all these relations in a producer metric the results could be used to calculate all the **package**

metric[7] metrics, avoiding the need to duplicate the relationship calculation for each metric.

Implementing the package metrics is a two part process. First the common data source needs to be defined, providing all the data that is defined by the paper. This includes sets of packages and classes that use and extend the target package, as well as the reverse-mappings thereof. The data required to calculate this is gathered by a shared producer metric, which creates a data object for each package, representing the relations of that package.

For the implementation of the package metrics, the paper makes a distinction between a class that **uses** another class and a class that **extends** another class. The given definition is that a class **extends** its direct superclass and a class **uses** another class if a field or method in that class is accessed and that class is not a superclass of the inspected class. The full list of sets is listed below.

```
PackageUnit:
- getPackageName() //Identifier of the package
- getPackageUnit(packageName) //Get Unit for other package
- Int() //All classes that use/are used by external packages
- OutInt() //All classes that use external packages
- InInt() //All classes that are used by external packages
- ClientsP() //All packages that depend on this package
- ProvidersP() //All packages that this package depends on
- ClientsP(className) //All packages that use the specified class
- ProvidersP(className) //All packages that are used by the specified class
- ClientsC() //All external classes that use classes in this package
- ProvidersC() //All external classes used by classes in this package
- Uses() //Set of all packages USED by this package
- UsesC() //Set of all classes USED by this package
- Ext() //Set of all packages EXTENDED by this package
- ExtC() //Set of all classes EXTENDED by this package
- UsesSum() //Set of all classes USED by this package, including internal use
- ExtSum() //Set of all classes EXTENDED by this package, including internal extending
```

Creating this list of predicates was the most time consuming part of the process, since the definitions were not always clear. The issues encountered will be discussed further in subsection 4.2.2.

Given the data provided by the **PackageUnit**, implementing most metrics was simply a case of rewriting the set theory used to define it using the sets given by the predicates listed above. The **Guava** utility **Sets.intersection(Set, Set)** is used a lot to accurately implement the set intersections defined by the paper.

In order to ensure the results are correct, it is important to validate the implementation through testing. This is done for the package metrics by turning all examples given in the paper into **Unit tests**[8], and running the entire test suite during the building process.

Since the package metrics are defined at a package level, creating the test cases involved creating a set of classes with the same relations among them as the examples given in the paper, the classes were not required to be functional or useful, making it easy to create a large set of test packages in a short amount of time.

Given the data provided by the **PackageUnit** and the validation provided by the **unit tests**, a correct implementations for all the package metrics were found. The implementations also share names and structure with the formulas defined in the paper, adding a way to validate their correctness. An example implementation can be seen at Listing 4.5.

[7] Abdeen, Ducasse and Sahraoui's metrics are named 'Package Metrics' internally due to lack of a better name

[8] A software testing method that focuses on testing small parts of source code for correctness.

Listing 4.5: nl.rug.jbi.jsm.metrics.packagemetrics.PF

```
39  @Subscribe
40  @UsingProducer(PackageProducer.class)
41  public void onPackage(final MetricState state, final PackageUnit
        pack) {
42      state.setValue("Collection", pack.getSourceIdentifier());
43      state.setValue("PF-p", calcPF(pack));
44  }
45
46  private double calcPF(final PackageUnit pack) {
47      final Set<String> usedPackages = pack.ClientsP();
48      if (!usedPackages.isEmpty()) {
49          double res = 0.0;
50          for (final String pi : usedPackages) {
51              final PackageUnit pu = pack.getPackageByName(pi);
52              res += Role(pack, pu);
53          }
54          return res / usedPackages.size();
55      } else {
56          return 1.0;
57      }
58  }
59
60  private double Role(final PackageUnit p, final PackageUnit q) {
61      final double InIntPQ = Sets.intersection(p.InInt(), q.ProvidersC
            ()).size();
62      return InIntPQ / p.InInt().size();
63  }
```

4.1.6 *Dealing with Exceptions*

During the evaluation of metrics, the tool will encounter exceptions caused by the way the metrics are defined. Implemented metrics are software and software is almost never bug free. The tool needs to make sure it can safely execute the calculations and that exceptions do not cause incorrect results.

During registration the tool needs to make sure that the metric is defined correctly before actually registering the metric. Below are some of the checks performed before completing registration.

- The metric extends one of the accepted base classes, **IsolatedMetric** or **SharedMetric**. If this is not the case the system would not be able to perform the finalization.
- The results of the metric need to be of the same or higher scope than the metric itself. If a **package** scope metric produces results for the **class** scope, it violates the scope sequence required by the execution plan. This check includes producer metrics since their produce is subject to the same constraint.
- Make sure that there is a frame within the execution plan that has all the data required by the metric, so the metric can be finalized in that frame. Missing data indicates the metric expects data that does not exist or expects the wrong data.
- Make sure any method annotated with the **@Subscribe** annotation has the correct method signature. (*public void method(MetricState,DataType);*).
- If a method has the **@UsingProducer** annotation, make sure the expected data type matches the data produced by the producer metric.
- Make sure the producer is registered if a metric indicates it requires the producer, failing if the registration of the producer fails.

Failure to pass all of these checks will cause the tool to throw an exception and shut down after notifying the user of the exception.

Next it is possible the metric is correctly defined, but that they throw an exception during calculation. These exceptions are handled differently depending on the place the exception is thrown.

If an exception is thrown by the data handlers, the state for the target in question is marked as invalid. This is because the state cannot be considered correct after throwing an exception while it was being changed. If the metric that the state belongs to is an **IsolatedMetric**, then it is impossible to calculate a result for that target and the system will send an **InvalidResult** object instead of asking the metric to calculate a result.

If the metric is not an **IsolatedMetric**, it is a **SharedMetric** or a **ProducerMetric**. Since the results for these metrics are dependent on more data than just the data from the failed target, the finalization methods include an argument in which the system passes the number of targets for which an exception occurred during data gathering.

It is up to the developer implementing these metrics to decide whether this causes the results to be useless and abort the calculation. The system will pass all successfully gathered data to the metric regardless of the number of targets for which the gathering failed.

The next place an exception can occur is during finalization of a metric. Exceptions thrown by the finalization methods are caught and logged so the user can determine and fix the cause of the exception.

Since metrics are executed and evaluated in parallel, it is extremely important that exceptions caused by one metric do not disrupt the calculation of other metrics. The tool makes sure that this cannot happen by performing all invocations of the metric calculators in **try catch** blocks, catching exceptions thrown by the metric.

The thread executing the calculations has a special **UncaughtExceptionHandler** to catch uncaught exceptions. This had to be done because the calculations are performed asynchronously, so if an uncaught exception occurs the processing thread would never signal that it was finished to the front-end, leaving it waiting for that signal until the tool is killed.

Any failure during calculation is logged for debugging purposes. The GUI contains a console that shows the exception message but does not show the exception stack trace to keep the console readable. Since the exception stack trace is invaluable for debugging the cause of the exceptions, it is written to a log file for future inspection.

4.2 Issues during implementation

While creating the implementation of the design described in chapter 3, a number of issues were encountered. Most of these issues had to do with the definition of the metrics that had to be implemented by default in the tool. This section aims to explain the background of these issues and how they were solved.

4.2.1 *Assumptions in CKJM*

One of the ways types can be used in Java is through local variables. This means that to build the full list of used types required by CKJM's **CBO** and **CA** metrics, these types need to be tracked. The reference implementation does this by catching the instructions shown in Listing 4.6, Listing 4.7 and Listing 4.8.

Listing 4.6: gr.spinellis.ckjm.MethodVisitor

```
73 /** Local variable use. */
74 public void visitLocalVariableInstruction(LocalVariableInstruction i)
       {
75     if(i.getOpcode() != Constants.IINC)
76         cv.registerCoupling(i.getType(cp));
77 }
```

Listing 4.7: gr.spinellis.ckjm.MethodVisitor

```
79 /** Array use. */
80 public void visitArrayInstruction(ArrayInstruction i) {
81     cv.registerCoupling(i.getType(cp));
82 }
```

Listing 4.8: gr.spinellis.ckjm.MethodVisitor

```
110 /** Visit return instruction. */
111 public void visitReturnInstruction(ReturnInstruction i) {
112     cv.registerCoupling(i.getType(cp));
113 }
```

The issue with this way of gathering the types is that they do not work as expected in BCEL. All of the calls to **getType** in the above shown code only return basic types based on the opcode of the instruction, not the type of the local variable they are operating on.

What this means for the reference implementation is that it does not correctly show types used by local variables in instances where a field or method of that type is not invoked. In practice the definition of a local variable implies the methods or fields of that variable will be accessed, meaning the use of the type is registered.

In order to correctly support the discovery of local variable types, the **LocalVariableTable** should be used. This set of data is stored in the bytecode of the class and when a method uses a local variable, it refers to this table. The default BCEL class visitor exposes the data of the local variables through the code shown in Listing 4.9, correctly exposing the actual types of the local variables.

Listing 4.9: nl.rug.jbi.jsm.bcel.MethodVisitor

```
57 public void visitLocalVariables(final LocalVariableGen[] lVarGens) {
58  logger.trace(Arrays.asList(lVarGens));
59
60  if (!this.getEventBus().hasListeners(LocalVariableDefinition.class))
        return;
61
62  for (final LocalVariableGen lVarGen : lVarGens) {
63      this.getEventBus().publish(new LocalVariableDefinition(lVarGen))
            ;
64  }
65 }
```

4.2.2 *The package metrics specification*

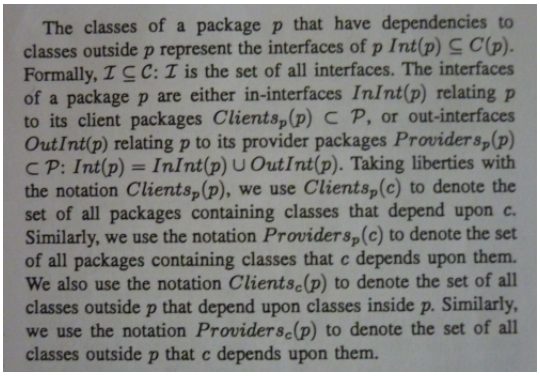
When looking at the time spent implementing the metrics, the vast majority of it was spent working on the package metrics. This had a few reasons but chief among those reasons were the way the metrics were defined, as well as the fact that the paper was incorrect in places.

Due to the fact that the public paper by Abdeen, Ducasse and Sahraoui [1] is a publication, it is not the full version of the paper. For the purpose of this project a full version of the paper was available for use, but due to the fact that it had not been edited for publication it was harder to read than a publication would have been.

The paper defines metrics for what it calls a **Modularization**. This is a set of packages and the 'Uses' and 'Extends' relationships between those packages. To define metrics about these relationships, the paper defines a set of predicates that operate on individual packages[9].

The issue begins with the way these predicates are defined, displayed in Figure 11. Due to the way the text is structured in the paper, it becomes incredibly difficult to determine exactly what the definition of a predicate is and what it represents.

[9] A list of predicates can be found in the metric implementation section



The classes of a package p that have dependencies to classes outside p represent the interfaces of p $Int(p) \subseteq C(p)$. Formally, $\mathcal{I} \subseteq \mathcal{C}$: $\mathcal{I}$ is the set of all interfaces. The interfaces of a package p are either in-interfaces $InInt(p)$ relating p to its client packages $Clients_p(p) \subset \mathcal{P}$, or out-interfaces $OutInt(p)$ relating p to its provider packages $Providers_p(p) \subset \mathcal{P}$: $Int(p) = InInt(p) \cup OutInt(p)$. Taking liberties with the notation $Clients_p(p)$, we use $Clients_p(c)$ to denote the set of all packages containing classes that depend upon c. Similarly, we use the notation $Providers_p(c)$ to denote the set of all packages containing classes that c depends upon them. We also use the notation $Clients_c(p)$ to denote the set of all classes outside p that depend upon classes inside p. Similarly, we use the notation $Providers_c(p)$ to denote the set of all classes outside p that c depends upon them.

Figure 11: Unclear definition of package relations

What actually ended up happening during development is that the definition and meaning of each predicate was determined by its use in the actual metrics. By defining the metrics using undefined predicates, the content and definition of those predicates could be reverse engineered from the paper explaining what the metrics were supposed to calculate.

However to ensure that the results are correct, these definitions need to exactly match the implied definitions in the paper. The definitions were validated by writing unit tests for all of the examples given in the paper.

This is also where the fact that the paper was wrong was discovered. In the third example given for **IPSC**, the correct value according to the paper should be $\frac{32}{42}$. This is incorrect, as the following calculation will show:

$$
\begin{aligned}
cohesion(p, p1) &= \frac{\frac{8}{5}}{3} = \frac{8}{15} = \frac{432}{810} \\
cohesion(p, q2) &= \frac{\frac{5}{2}}{3} = \frac{5}{6} = \frac{675}{810} \\
cohesion(p, q3) &= \frac{\frac{5}{3}}{3} = \frac{5}{9} = \frac{450}{810} \\
IPSC(p) &= \frac{\frac{432+675+450}{3}}{810} = \frac{519}{810} = \frac{173}{270}
\end{aligned}
$$

This shows that the correct result for example 3 of **IPSC** is $\frac{173}{270}$, not $\frac{32}{42}$.

Besides this case of a wrong example, all of the other examples given by the paper were correct. By turning these examples into a set of unit tests a test suite is created that validates the implementation of the predicates and metrics by comparing the results to the results in the paper.

By using these tests to ensure the results matched the examples given in the paper, all the required predicates were defined and implementation could be completed.

5 EVALUATION AND RESULTS

In this chapter the tool will be evaluated by comparing its output to the output of the reference implementations. By ensuring the output is the same the correctness of both implementation of metric and execution is proven. Besides the implementation validation this chapter includes a report on the performance of the tool.

All evaluation will be performed in the following environment:

```
Kernel: i686 Linux 3.14.6-1-ARCH
CPU: Intel Core2 Duo CPU T5750 @ 2GHz
RAM: 3028MB
Java version: "1.8.0_05"
JSM release: 1.2
CKJM version: 1.9
```

5.1 COMPARISON TO REFERENCE CKJM IMPLEMENTATION

Since the implementation of the C&K metrics is based on CKJM, it is fair to compare JSM to that implementation in terms of correctness of results, performance and usability.

5.1.1 *Correctness of Results*

This test will be performed by evaluating the JSM binary JAR file with both JSM and CKJM. The results from both evaluations will then be compared and should be exactly the same if both implementations are correct.

The CKJM binary is invoked with the following script:

```
1 #!/bin/bash
2
3 jar tf jsm-1.2.jar |
4 sed -n "/\.class$/s,^,jsm-1.2.jar␣,p" |
5 java -jar ckjm-1.9.jar |
6 ./ckjm2csv > ckjm_result.csv
```

This script was written based on the CKJM documentation found at `http://www.spinellis.gr/sw/ckjm/doc/insel.html` and `http://www.spinellis.gr/sw/ckjm/doc/outfmt.html`.

JSM will be invoked with the following script:

```
1 #!/bin/bash
2
3 java -jar jsm-1.2.jar \
4     --in jsm-1.2.jar \
5     --out results/jsm.%s.csv \
6     --group-scopes
```

This script is based on the JSM scriptable documentation found at `https://github.com/Kiskae/Java-Source-Metrics/blob/master/HOWTOUSE.md#script-mode`.

By importing the results into Excel, checking for equality between the CKJM and JSM results for each metric and then counting all cases where the results do not match, we get the following table:

	NOC	CBO	RFC	LCOM	CA	WMC	NPM	DIT
Mismatches	0	1	1	0	1	0	0	1060

Perhaps unexpectedly, there are deviations between the results. These deviations can be explained by looking at the differences in implementation between CKJM and JSM.

First the different results for **CBO** and **CA**, due to the fact that CA is the reverse mapping of CBO these differences have the same cause. This cause is the assumption made by the CKJM developer discussed in subsection 4.2.1. In the Guava library, there is one class that uses a class in a local variable that is not used in any other way. This means that CKJM was not able to detect the use, but JSM was.

The reference that causes the mismatch of one reference was between *com.google.common.base.Suppliers* and *com.google.common.base.Suppliers$SupplierFunction*, as shown below in the disassembled output of *com.google.common.base.Suppliers*.

```
LocalVariableTable:
 Start  Length  Slot  Name   Signature
     4       2     0    sf   Lcom/google/common/base/Suppliers$SupplierFunction;
```

The cause of the mismatched results in **RFC** was harder to find, but is actually caused because CKJM uses a deprecated[1] method in BCEL. The deprecated method is **InvokeInstruction.getClassName(ConstantPoolGen)**, which has a note in the documentation saying the following:

```
*  @deprecated If the instruction references an array class,
*    this method will return "java.lang.Object".
*    For code generated by Java 1.5, this answer is
*    sometimes wrong (e.g., if the "clone()" method is
*    called on an array).  A better idea is to use
*    the getReferenceType() method, which correctly distinguishes
*    between class types and array types.
```

This warning was noticed during development of the JSM tool, so the advice given in the warning was applied in the JSM implementation of **RFC**.

The fact that CKJM uses the deprecated method caused it to count the invocation of **.clone()** on two different types of arrays as a single invocation of **Object.clone()**. While JSM considered these calls two separate instances of **.clone()**.

In the decompiled output of *org.apache.logging.log4j.core.pattern.AnsiEscape* the two calls to **.clone()** can be seen. CKJM sees these as a single call, while JSM considers them separate, causing the mismatch in results.

```
#2 = Methodref // "[Lorg/apache/logging/log4j/core/pattern/AnsiEscape;"
                                                .clone:()Ljava/lang/Object;
#20 = Methodref // "[Ljava/lang/String;"
                                                .clone:()Ljava/lang/Object;
```

The final set of differences in results are the results for **DIT**. While the number of mismatches seems to be worryingly high it seems to be caused by a problem with the execution environment. All of the mismatches were cases where the results for CKJM were 0, which is not a valid result for **DIT**. However, CKJM will return a result of 0 if it fails to be able to find all the superclasses for a class.

Looking at the implementation, the mismatches can be considered void since the implementation of **DIT** is exactly the same in both CKJM and JSM. The difference lies in the fact that JSM defines a custom **Repository** to perform class resolving, while CKJM relies on the default repository.

The default repository in BCEL loads classes from the classpath, since the JAR files from which the classes originate are not on the classpath CKJM will fail to load any class that is not a part of the Java API or included in CKJM's binary.

Because both JSM and CKJM use BCEL, all BCEL classes and Java API classes will be resolved by CKJM, but classes that extend any class not part of that set will cause the **DIT** calculation to fail.

[1] Deprecation is a term in software development that indicates a feature should be avoided

5.1.2 *Performance Comparison*

For the purpose of this test the JSM tool has been modified to disable the package metrics, since their removal means both JSM and CKJM are calculating the same results.

Both tools will be given a .jar and their runtime will be measured for comparison using the bash tool **time**. The tools will be executed in the exact same way that they were during the correctness section.

The results can be seen in the table below.

Input	Number of Classes	CKJM Runtime	JSM Runtime
Empty JAR file	0	0m0.178s	0m4.686s
JSM Binary	2769	0m8.632s	0m12.327s
IntelliJ idea.jar	30662	5m0.249s	2m0.359s

What these results show is that JSM requires more time to begin execution, as shown by the longer runtime for a small set of classes. However the parallel calculation of metrics means that JSM scales much better when given a very large set of classes, being multiple minutes quicker than the CKJM implementation for the application with 30.000 classes.

5.1.3 *Usability*

Comparing usability between CKJM and JSM is rather one-sided because CKJM is meant to be a simple unix utility. The inclusion of the GUI makes JSM more usable for anyone not familiar with the command-line. The fact that JSM is not designed to be a unix utility means it can be easily used on other platforms, which is not the case for CKJM.

The most obvious difference in usability for a developer is the logging. CKJM only outputs logging if an exception occurs. If BCEL is unable to find the superclass definition for a class it logs the string representation of that class to the command-line, resulting in a lot of noise in the debugging output.

JSM logs a lot of useful information for the developer, allowing the user to track how the calculation progresses. If an unexpected exception occurs, it will log it to the console with a message specifying where it failed. The BCEL class visitor includes a lot of tracing statements, making it possible to have JSM output all the BCEL data to the console, but these statements are disabled by default due to their verbosity.

5.2 PERFORMANCE

When it comes to performance, the most important metric is the amount the tool scales relative to the input.

The performance measurements are performed by submitting a number of different Java applications to be evaluated by JSM. By choosing applications with different amounts of classes, a small sampling of the scalability of the tool can be performed.

Execution time is measured using the bash utility **time**, which measures the time taken from invocation of the program until its termination. JSM is executed in script mode and the elapsed runtime is measured with **time**, the number of classes in the resulting .csv files give a count of the classes in the inspection.

The results from the test are shown in the table below, as well as a plot in Figure 12.

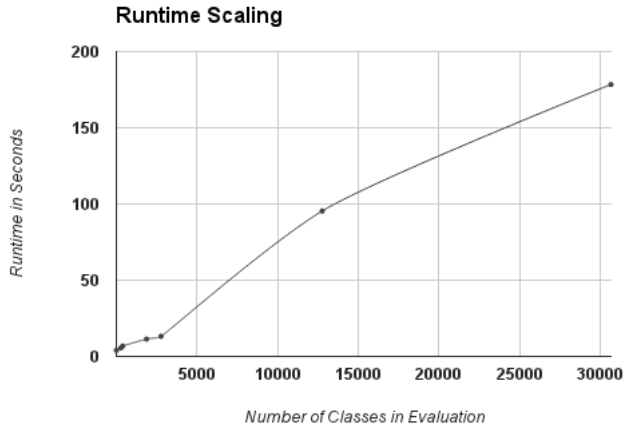


Figure 12: Runtime scaling of the tool

Input	Number of Classes	Runtime
Empty JAR file	0	0m3.978s
jUnit and Hamcrest	278	0m5.535s
CKJM binary	392	0m6.796s
Minecraft Client	1869	0m11.371s
JSM Binary	2769	0m13.170s
Minecraft Server	12761	1m35.388s
IntelliJ idea.jar[2]	30662	2m58.389s

These results paint a good picture for the scalability of the tool with runtime scaling linearly with the number of classes that need to be evaluated. This is an ideal situation for this tool considering the nature of its work. A larger number of classes as input naturally causing a linear increase of work to evaluate all those classes.

While performing the tests, some observations about the behaviour of the tool could be made. The tool takes about 4 seconds to start up regardless of input, as shown by the empty JAR file test case. Judging by the output of the tool during this time, those 4 seconds are spent initializing the logging system and registering the metrics.

Another observation about the runtime behaviour of the tool is that the pause between accepting the input and beginning calculation scales linearly with the amount of classes in the input. This seems to be caused by the fact that all classes need to be loaded and parsed by BCEL before calculation can begin, an activity that can only be done sequentially due to the Class Loader. Deferring the loading of these classes until they are used might be a way the tool can be sped up in the future.

The memory issues encountered in the largest test-case seem to indicate there might be a memory-leak in the way the tool caches class data. This seems to be due to the way BCEL handles caching in its repository and is a possible method of improvement for the tool in the future.

[2] To handle idea.jar, the heap had to be set to 2gb to allow all classes to load.

6 CONCLUSION

The main goal of this project was to design and implement an extensible metric calculation tool for Java. The object-oriented metrics defined by *Chidamber and Kemerer* and the modularity metrics defined by *Abdeen, Ducasse and Sahraoui* had to be implemented as metrics within this tool.

By looking at the way metrics are defined, two metric archetypes were defined with an additional type that bridges the system-provided data with user-defined data. These types respectively are the **isolated**, **shared** and **producer** metrics.

These archetypes were then translated into abstract Java classes, exposing methods that perform the metric finalization step of turning gathered data into results. Through the use of Java annotations, an event-based handling system for the data used by the metrics was created. The handler uses the method signatures as registration for an EventBus system through which metrics can define their interest for data exposed by the system, or user-defined **Producer** metrics.

By building a directed acyclic graph of metrics using the dependencies caused by the **producer** metrics, an order of execution is determined by the **Pipeline Frames** system. This system acts as a validation of the metric definition as well as a guarantee that calculation will end. By finding the frame in which all data has been delivered to the metric, the frame in which a metric can be finalized is determined. This data is used to define an execution plan for the execution system.

The execution system heavily uses the metric archetypes and the pipeline frames execution plan to optimize the calculation of the metrics. Through the use of subtasks as much work as possible is done in parallel whilst maintaining the thread-safety of the calculation, and the sequence of execution defined by the execution plan. When the execution plan indicates a metric is finished, the results are finalized and sent back to the user.

Once the tool was working the reference implementation of *Chidamber and Kemerer*'s metrics, CKJM, was modified to fit the metric archetypes. During this process several flaws in the implementation were found and amended in the **JSM** version.

Since there was no reference implementation of *Abdeen, Ducasse and Sahraoui*'s modularity metrics, they were implemented by turning the definitions of the paper into implementations based on the metric archetypes. The predicate data was pre-calculated using a **producer** metric. Due to the confusing way the paper defined the predicates it used in its metrics, the predicates had to be reverse-engineered. Correctness of implementation was asserted by creating unit tests based on the examples given in the paper. These tests also greatly helped the reverse-engineering effort.

Through the use of the metric archetypes and the 'pipeline frames' metric sequencing system, the project is a success in creating an extensible metric calculation tool. Using tests the metric implementations provided with the tool were validated, asserting they return the correct results for their input.

Personally I believe that through the definition and use of the metric archetypes, the system should be able to handle any arbitrary metric that can access the data it requires through BCEL and follows the definition of a metric archetype. This is the definition of an **extensible** system. The fact that much of the calculation done by the executing system is performed in parallel implies the way the metrics are defined is ideal for calculation purposes.

The full source code as well as binary releases can be found licensed under the MIT license at `https://github.com/Kiskae/Java-Source-Metrics/`.

6.1 LIMITATIONS

While the runtime performance and the ability to create new metrics based on the archetype are great for extensibility, there is another method of extending the tool. This is by replacing the Class Visitor with a custom version, emitting data in addition to the base data. This type of extension is badly supported by the way the tool is implemented, requiring the user to modify two components before it will work.

Another limiting factor for the tool is the focus of the default BCEL class visitor and scope system. It is designed to expose all data related to relationships and communication between classes, but lacks exposure of details for other purposes. This makes implementing metrics that focus on smaller pieces of an application extremely difficult. It would be very inconvenient to implement McCabe's Complexity Metric due to the lack of information on branching instructions exposed by the default implementation, as well as the fact that the branching instructions are more ambiguous than source code would be.

Finally the way metrics are defined is not strictly Object-Oriented. The metrics objects have the single purpose of being given a state by the system, modifying that state based on data exposed by the system and then creating a result based on that state. The strict avoidance of state in the metrics means they are designed to act like a function would in a functional programming language. While this way of defining metrics means it is incredibly easy to move data between calculation steps, it is not a design most Java programmers are familiar with.

6.2 FUTURE WORK

One of the first major areas of focus when improving the tool would be the way BCEL class data is loaded and stored. Because the tool currently loads all classes that need to be evaluated before it begins calculating the metrics, all class data needs to be kept in memory for most of the execution. This causes the tool to show issues related to memory leaking when the input starts getting larger. By deferring data loading until the data is required this issue could be avoided.

Another way the tool can be expanded upon is to expand the set of data exported through the default BCEL class visitor. This would allow for the creation of other types of metrics without having to override the class visitor, which can only be done once. Based on these new types of exposed data, additional default metric implementations could be added to the tool.

Lastly the issue with extending the tool through a custom class visitor, mentioned in the Limitations section, could be solved by reimplementing the way the class visitor factory and default data-set are determined.

[1] H. Abdeen, S. Ducasse, and H. Sahraoui. Modularization metrics: Assessing package organization in legacy large object-oriented software. In *2011 18th Working Conference on Reverse Engineering (WCRE)*, pages 394–398, October 2011.

[2] P. Avgeriou and Uwe Zdun. Architectural patterns revisited - a pattern language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, pages 1–39, Irsee, Germany, July 2005.

[3] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[4] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[5] D. Owens and M. Anderson. A generic framework for automated quality assurance of software models - application of an abstract syntax tree. In *Science and Information Conference (SAI), 2013*, pages 207–211, Oct 2013.

[6] Diomidis Spinellis. Tool writing: A forgotten art? *IEEE Software*, 22(4):9–11, July/August 2005.