Code Maintainability Analyzer Operational Concept Diagram (OCD)

Mudit Vats
07/22/2018

Table of Contents

Table of Figures

Acronyms

This section describes acronyms used throughout the document.

CMA	Code Maintainability Analyzer
LCOM	Lack of Cohesion of Methods
LOC	Lines of code
MI	Maintainability Index
OCD	Operational Concept Diagram
SQA	Software Quality Assurance
GUI	Graphical User Interface

Executive Summary

The Code Maintainability Analyzer (CMA) application is an application which can be used to understand the health of a set of source code. It does this by calculating metrics on the source code and sharing the results with the end-user. The results can be one of several metrics and a final overall index that can be used to quickly gauge the quality (i.e. maintainability) of the software.

Why do we need this kind of application? Consider the following uses –

- A Software Quality Engineer may want to understand the health of a set of source code before the software is being released.
- A Developer who wishes to understand if the code he or she is producing is too complex or poorly organized. Along the same lines, a Developer who is code reviewing someone else's code may wish to understand the same things.
- A Software Architect who isn't working on the actual code but is interested in ensuring that the code is well constructed and can be maintained.
- A Software Quality Assurance engineer who needs software metrics to ensure the quality of the software is tracking to the standards the program expects.
- A Validation engineer who wishes to understand which areas of software need the most amount of testing. For example, areas with large complexity may need more testing to ensure all code paths are validated.

When considering these uses (and others), we can see that there are several reasons why having an application which can give metrics on the source code can benefit a software development team.

The key ideas to take from this document include –

- This document describes a C# .Net console-mode application called the Code Maintainability Analyzer application which delivers metrics to the end-user for a set of source code.
- The metrics delivered are Lines of Code (LOC), Complexity, Cohesion, Coupling, and Maintainability Index (MI).
- The design is decomposed into Activities, grouped into Partitions and described via class diagram(s). This document contains the details which should allow this project to move to the next phase of software development. In our case this is Project 2, which is the implementation of the CMA application.
- The design of the CMA is a logical extension of the Code Analyzer project. As such and in consideration of the architecture described in this OCD, it is believed that the goals of this project are completely feasible to develop in a *reasonable* amount of time. For this project, that's three weeks for development and test.

Critical Issues with mitigations are explored and discussed in Critical Issues section. Some of the high-priority issues worth attention are -

- Single Threaded Application Design
- Identifying Non-System Dependencies
- Intention vs Metrics
- Future Proofing
- Extreme Input Load

The rest of this document delves into the architecture by presenting uses, activities and partitions.

Introduction

Obligations

The Code Maintainability Analyzer Application gets a list of files from the end user and produces metrics which the end-user can use. It should provide the results to the console, or if the user desires, to a file. Additionally, and most importantly, the CMA should compute metrics for Lines of Code (LOC), Complexity, Cohesion, Coupling and, finally, a Maintainability Index.

Organizing Principles

At a high-level, the architecture of the CMA application consists four functional areas:

- Command-line Processor
- Parser
- Compute Metrics
- Results Logger

The Command-line Processor validates and makes the command line arguments the user specifies to the rest of the application. Parser, parses the source code files to understand types and class relationship information. It uses Rules and Actions which describe what to look for in the expressions being parsed and actions which dictate what to do with that type. Compute Metrics computes the metrics by gathering the relevant data from the parser results in the Repository. Finally, the Results Logger logs the results for end-user analysis. The details of these packages and the activities which help define the packages are discussed in subsequent sections of this document.

Key Architectural Ideas

- The CSharp_Analyzer project and several contained packages are reused and/or extended.
- The Rules and Actions package will be extended to handle discovering class relationships, such as those used in Coupling and Cohesion.
- The CMA application will be command-line only. No GUI.
- The Parser will consist of a two-pass approach. This is necessary so that after all of the names/types are identified for all classes in all files (first pass), then the class relationships can be understood (second pass).

- The design of the CMA application is *purposely* meant to easily allow the addition of new metrics in the future. For example, to add a new metric, we can simply add more rules and actions to the Parser. This allows us to identify new expression types and add them to our parsing results. Similarly, we can extend Compute Metrics to compute the new metrics.

Uses

Generally, it is desirable for anyone working in software development to understand the quality of the source code they're working with. The CMA application allows this capability by computing the LOC, Complexity, Cohesion, Coupling and Maintainability Index. By using this application, developers, architects, managers, quality assurance, validation engineers, program managers and managers have a go-to method for gathering specific measurable insights into the software they're interacting with.

Before considering the uses, below are the definitions of the metrics that are computed by the CMA application.

Metric	Definition
LOC	Lines of code for a function, class or namespace.
Cyclomatic Complexity	The number of independent paths in a function.
Cohesion	Lack of cohesion in methods (LCOM1). LCOM1 measures the cohesion in terms of the number of pairs of functions that do not share any data members.
Coupling	A measure of the dependency between classes. A class A depends on class B if A inherits, aggregates, composes, or uses class B.
Maintainability Index	Index defined by the sum of all metrics. MI=a * LOC + b * Cyclomatic_Complexity + c * Cohesion + d * Coupling Where a, b, c, d are user-defined coefficients passed into the CMA application.

Figure 1: Metric Table

As you can observe, understanding these metrics can give a great deal of knowledge about the structure and maintainability of the software being measured. Some metrics are more or less valuable to the users who use them. We discuss how different users would use the CMA application and which metrics may be valuable to them.

Developers

Developers are the users most often to use the CMA application. This is mainly because they are the ones writing the code and have immediate need to understand the quality of the code they're developing.

A developers' needs are very specific and have the most granular expectations. Since they are actually working on the code, having the specific metric values can help them

fix, reorganize, optimize and, ultimately, produce higher quality code. They would run the CMA application on their source code iteratively as they develop the code.

For developers, all of the metrics are valuable for their scrutiny. Iteratively understanding how the new code they write affects these metrics can help ensure they are not introducing poorly organized code into the project's code base.

On the flip-side, a developer who is code reviewing someone else's source code may wish to understand the same metrics to better understand the complexities and organization of the code.

As previously mentioned, additional metrics can be added to the CMA application. As the code base grows, it should be easy to add more metrics to suit the teams' needs. This way, the CMA application can be integrated into the continuous integration system so that current metrics can always be made available and future metrics can be easily accommodated.

Software Architect

A Software Architect can be one or many individuals. They can also be developers, but usually they focus their abilities in understanding the subsystems making up system, how they are organized and how they communicate with each other. Still, while the Software Architect may not be involved in the day-to-day code development, they would still be interested in the actual quality and maintainability of the code which the software development team is producing.

Since the Software Architect is at the component level, he or she may be interested in the interdependencies within packages and amongst packages. This is where Cohesion and Coupling can be very valuable to the Software Architect. Cohesion is used to understand how methods within class are related by measuring functions which share attributes. When many methods share one more attributes, that's considered high cohesion – a situation where methods are related. When they don't share many attributes that would result in low cohesion and may suggest a re-architecture; i.e. splitting classes so that each class has common functionality and, ideally, create more cohesion within those classes.

Similar to Cohesion, a Software Architect may wish to understand Coupling between classes so that they can comprehend how maintainable the software is. For example, in CSE681, we learned about high coupling with the Mozilla project which resulted in one modification in one component significantly impacting other components. A Software Architect should be concerned with interdependency and making sure a low level of Coupling exists within the system so to ensure the source can be reasonably maintained in the future.

Of course, the Maintainability Index is the first stop in understanding the health of code. Since coefficients can be passed in to the CMA application, an Architect may weight metrics to suit his or her needs.

Quality Assurance

Quality Assurance Mangers and Engineers have their own motivations for understanding code metrics. Typically, QA has a set of criteria for Alpha, Beta, and Gold software

releases. These criteria may include the number of allowable defects, test code coverage, security testing (penetration testing, hackathons), code reviews, requirements traceability matrix et al. Amongst these factors, QA would use Maintainability Index to understand how well the code has been developed. Since the MI is a single metric encompassing LOC, Complexity, Cohesion, and Coupling, it can be used to gauge and judge software quality. A high MI value would indicate that the code may not be in the best shape, let alone, be a challenge to maintain. A low value would imply that new features and bug fixes can may be easily accommodated and that the overall health of the source code is "good".

Of course, a "good" value depends on the Software Quality Assurance team. Fortunately, the Code Analysis Application is built to handle different coefficients for each metric so that the MI is appropriately balanced based on the requirements set forth by the Software Quality Assurance team.

The QA team would present reports to upper management which would include the MI. They would get this measurement from nightly or weekly build reports.

Validation

Software Validation teams exist to create tests, execute tests, report defects and validate defect fixes.

During test development, using metrics provided by the CMA application such as Complexity can help validation engineers understand the more complex parts of the code. Additionally, Coupling can be used to understand inter code dependencies, which can help test designers target areas of code where the most interaction occurs. For example, if a handful of classes have a high coupling value, perhaps more tests should be developed to extensively validate those classes since a defect on one area can negatively impact other area as well.

Validation would also use the CMA application to ensure defect fixes are not introducing any new complexities which would require more tests to be developed. Of course, if that is the case, they would work with the software developer to either write more test cases or, perhaps, the software developer would need to redesign their fix.

Additionally, often times Validation is used as an independent source by Quality Assurance to provide an unbiased view of the health of the software. To that end, the Validation team would run the CMA application on currently developed code, perhaps third-party code, code from other organizations and provide those results to Quality Assurance.

Program Managers

Program Managers work with key stakeholders to, amongst other things, acquire resource for the program, help teams overcome obstacles, create project plans, and track program progress. From a software development perspective, a Program Manager would use the metrics produced by the CMA application to measure the quality and maintainability of code. Perhaps they may not be so interested in the individual metrics, but they would use the Maintainability Index to understand the overall health of the software.

A low Maintainability Index would indicate that the code produced by the development team is of higher quality; i.e. LOC, Complexity, Cohesion and Coupling are all at a level acceptable to the program. A high Maintainability Index would indicate the opposite and raise concern and follow-up by the Program Manager to discuss with the Software Development team to understand what's going on with the quality.

The metrics produced by the CMA application would be available to the Program Managers by reports via the nightly or weekly builds. They would use the MI, and possibly other metrics, to track progress and ensure the software trends to acceptable quality standards. Additionally, Program Manages would share this data with upper management and customers to keep them informed on the software development progress.

Manager

Managers may be interested code metrics the same way the Program Manager would. While the Manager may not own a product, he or she does own some of the resource working on the code, and, as such, has a vested interest in the success of the product. He or she may wish to use the Maintenance Index as a snapshot view into the health of the code being produced.

Software Managers usually have direct reports, who are Software Engineers; i.e. Developers. Managers would use the Code Analysis Application to understand how well their engineers are developing code. The Manager can take samples of the code one of his or her engineers works on and use the CMA application to produce the metrics to understand the quality of code their producing. The Manager may be interested in how the Developer is producing code over time. Upward sloping MI numbers may indicate the Software Developer may need training or guidance from a mentor to improve his or her skills.

Like the Program Manager, the Manager would pick-up the CMA application metrics from nightly or weekly build results. Sometimes, however, a Manger may not be subscribed to those reports since they may have many projects which would result in many reports in their inbox that are difficult to keep up with. For this situation a GUI to query results for a set of source code would be beneficial.

Application Activities

The diagram below represents a high-level overview of the Code Maintainability Analyzer application.

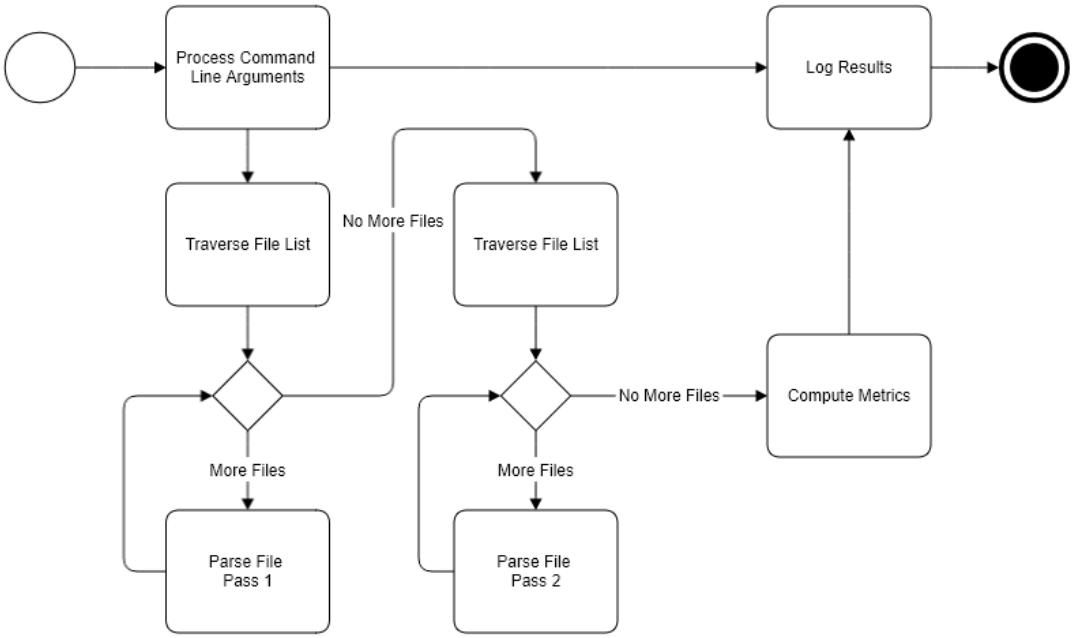

Figure 2: High-Level Overview

The major tasks of the application are listed below:

- Process Command Line Arguments
- Traverse File List
- Parse File Pass 1 and Parse File Pass 2
- Compute Metrics
- Log Results

The interaction with the CMA application starts with the end-user executing the CMA application. Various command line arguments can be specified to control the behavior of the application. Once the command line arguments are validated by the Process Command Lie Arguments task, the CMA application creates a list of files to process. The Traverse File List task begins by selecting each file and passing that along to the Parse File Pass 1 task. This is a serial operation in that one file parse occurs at a time. The idea of the parse operation is to deconstruct each file line into tokens so that the relevant data may be identified and classified and saved to the repository. Once all files are parsed and similar to File Parse Pass 1, File Parse Pass 2 again traverses the entire file list and uses the results from Parse File Pass 1 to comprehend and save class relationship data to the repository. Once the parsing is complete, the Compute Metrics task can query the repository to compute the desired metrics and store them back into the repository. Finally, the Log Results task accesses the repository to output the final metrics.

Note, while we reuse the Parser from Dr. Fawcett, this design differs in that all parsing is completed prior to metric computation. Additionally, we are implementing a "two-pass" Parser which allows the first pass to gather the type information for all files and the second pass to gather the relationship data. This differs from the original parser design since that only parses the files once, since no class relationship data is needed. Below are some examples to better understand how the two-pass parser works.

Consider the case of Coupling and needing to understand classes which *Inherit from* other classes. To understand this, the Parse File Pass 1, would identify classes and the inheritance of that class. This would be achieved by creating a *rule* which would look for the colon and name (public class MyDerivedClass : BaseClass) in the class definition. If this rule is satisfied, the *action* would be taken to add that inheritance data to the repository. Upon the second pass of the parser, Parse File Pass 2, the BaseClass will be checked against the file list to determine if this is a non-system class; i.e. a class defined by the set of parsed source code. If so, then this is a coupling relationship we would capture in the repository. Note, we do not capture relationships for system defined classes / types.

Another example, in the case of Coupling again, is if we wish to understand the *Use* relationship. As in the previous example, the first pass will find all the usages of attributes and class methods, and in the second pass we determine if these uses are part of the set of parsed source files. If so, we capture the relationship in the repository.

We can see how using a two-pass parser allows us to understand a bit about all files and then again understand how the files relate to one another. The following sections further decompose the high-level tasks.

Process Command-Line Arguments

The CMA application starts by handing off control to the Process Command-Line Arguments activity. This activity is responsible for validating command line arguments given by the user. The command line arguments specify various parameters and options which the user can specify to customize their experience with the CMA application.

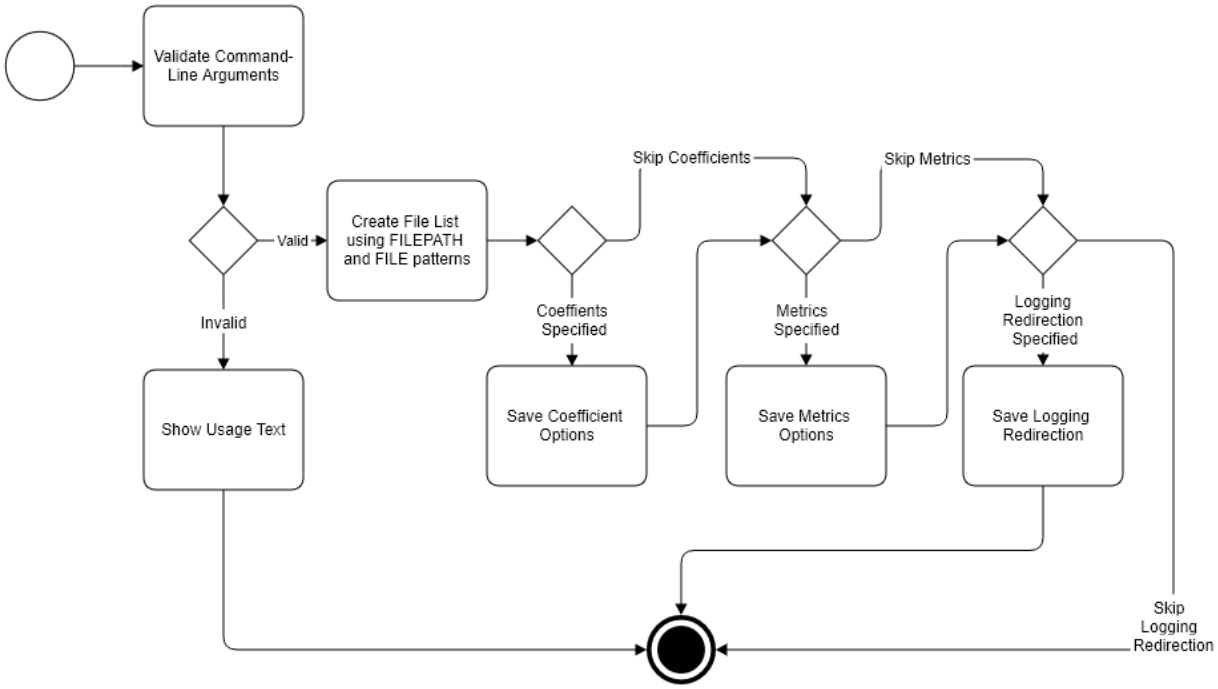

Figure 3: Process Command-Line Arguements Activity

As can be seen in the figure, command-line arguments are first validated by the Validate Command-Line Arguments task. If any of the command line arguments are invalid or if the required parameters are not specified, the Show Usage Text task executes and then terminates. If the arguments are valid, this means that we have a FILEPATH and some FILEs, then a file list is created by the Create File List using FILEPATH and FILE patterns task. Following this, if coefficients, specific metrics and logging options are specified, those options are saved by the Save Coefficients task, Save Metrics Options task and Save Logging Redirection task, respectively. Thereafter, we terminate.

The application usage is below:

- analyzer.exe FILEPATH FILE… <OPTIONS>

Parameters (FILEPATH, FILE) are required, whereas, options (<OPTIONS>) are optional.

The parameters include:

- FILEPATH which allows the user to specify the directory where files will be searched.
- FILE... which allows a space delimited list of files to be specified. The asterisk "*" wildcard may be used to specify matching patters within the filename. For example, "*.cs" would include all C# files in the FILEPATH directory.

The options include:

- Code Metrics Options

- "-l, -loc" to compute Lines of Code (LOC) per function, class, namespace.
- "-cx, -complexity" to compute code complexity per function.
- "-ch, -cohesion" to compute per class Cohesion value using the LCOM1 algorithm.
- "-cp, -coupling" to compute the per class Coupling value.
- "-m, -maintainabilityIndex" to compute the per class Maintainability Index.

- Coefficients
  - "-coL" is the coefficient for LOC metric; default is 1.0.
  - "-coCx" is the coefficient for the Complexity metric; default is 1.0.
  - "-coCh" is the coefficient for the Cohesion metric; default is 1.0.
  - "-coCp" is the coefficient for the Coupling metric; default is 1.0.
- Logging Redirection
  - "-file <filename>" to redirect console logging to a file name <filename>.
- Help
  - "-h, -help" to show help / application usage.

Compute Metrics Activity

The Compute Metrics activity computes metrics depending on the command-line input specified by the end-user. Compute Metrics relies upon the parsing data being fully comprehended in the repository by File Parse Pass 1 and File Parse Pass 2. The figure below shows the series of tasks for the Compute Metrics Activity.

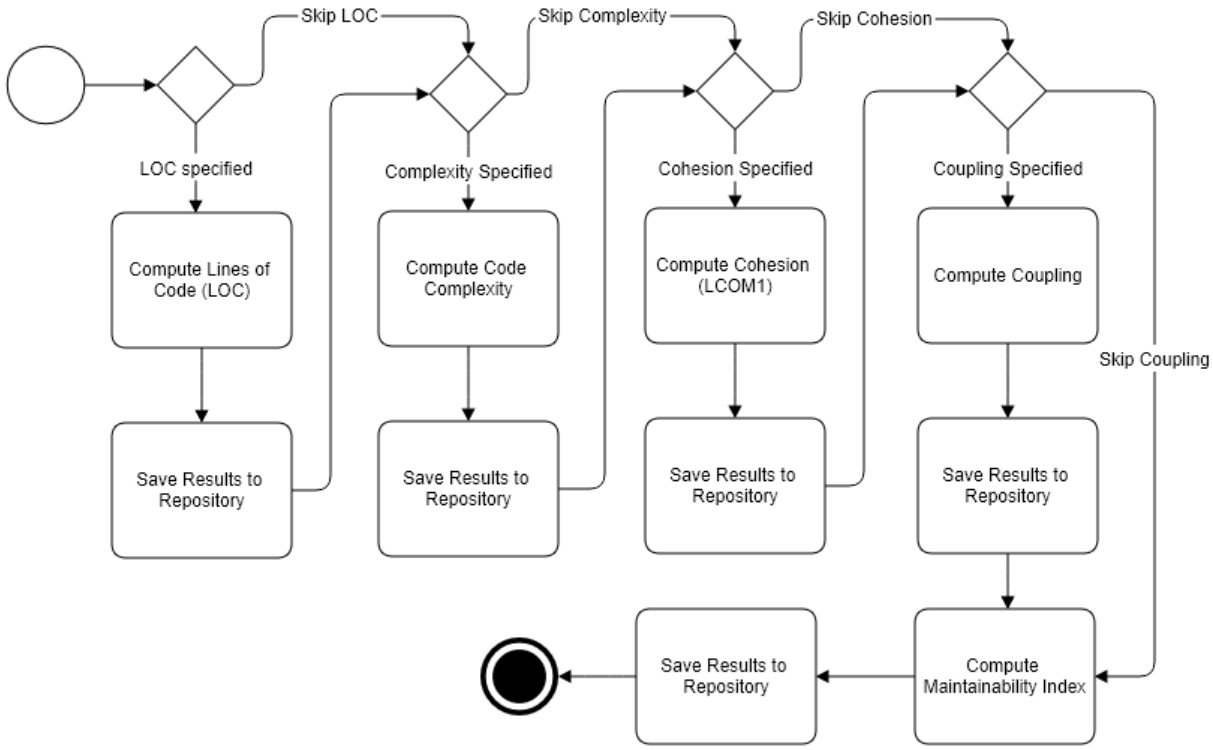

Figure 4: Compute Metrics Activity

The Compute Metric Activity follows the same pattern for each metric which can be computed. Based on command line input, Compute Lines of Code, Compute Code Complexity, Compute Cohesion and Compute Coupling may be executed. They all start by accessing the repository to get the parsed data results. These results are used in the computation of the final metric results. Once the metrics are calculated, their results are saved by the Save Results to Repository task. Finally, the Compute Maintainability Index executes and, just like the compute tasks, saves the MI result to the repository.

There are four standard metrics which can be computed, plus a final metric which is an index that includes the other four metrics. As previously discussed, the metrics are as follows –

- Lines of Code (LOC)
- Complexity
- Cohesion
- Coupling
- Manageability Index

The Compute Metrics activity computes the LOC and Complexity metrics using data previously generated by the File Parse Pass 1 task that is stored in the repository. For example, the Lines of Code is computed by subtracting the end-line of a function by the start line of the function. Similarly, Complexity is computed by taking a difference of the being / end scope.

Cohesion is computed per class using the LCOM1 formula. This is the difference between the number of pairs of functions within a class minus the number of pairs of functions that share at least one variable. Using these pairing values which would be stored in the repository by Pass 2 of File Parse Pass 2, we can compute the Cohesion value for a class.

For coupling, similar to the other metrics, once the inheritance, aggregation, composition and coupling relationship data is stored in the repository, we can easily compute the coupling results. This follows a similar approach as Cohesion, where type / name data is stored in the repository after File Parse Pass 1 and relationship data is stored in the repository after File Parse Pass 2.

Finally, the Manageability Index is computed as a sum of all the metrics multiplied by each of their respective coefficients, as passed into the CMA application. If no values are passed in, 1.0 is used as the default value for each coefficient. Additionally, since the end-user can specify which metrics to compute, only those metrics are computed, so we don't waste cycles on metrics we don't need to compute.

Log Results Activity

The Log Results Activity iterates through the results in the Repository and based on the desired output, either outputs the results data to the console or to a file.

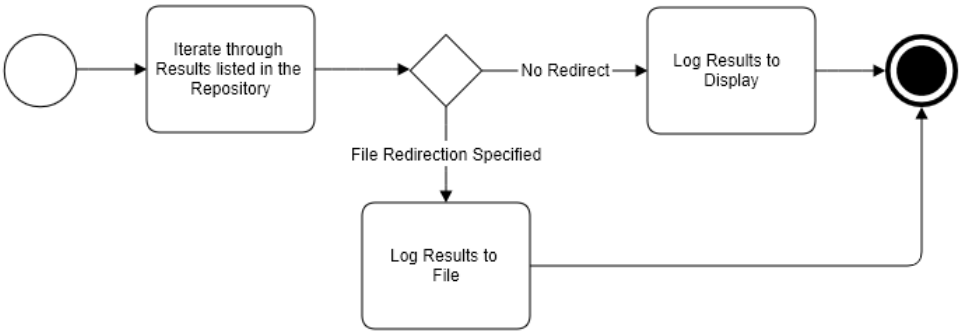

Figure 5: Log Results Activity

Partitions

This section shows the Package Diagram for the CMA application. Each package is broken-down into a brief description of the package, its responsibilities and interactions, including any local (non-system) dependencies.

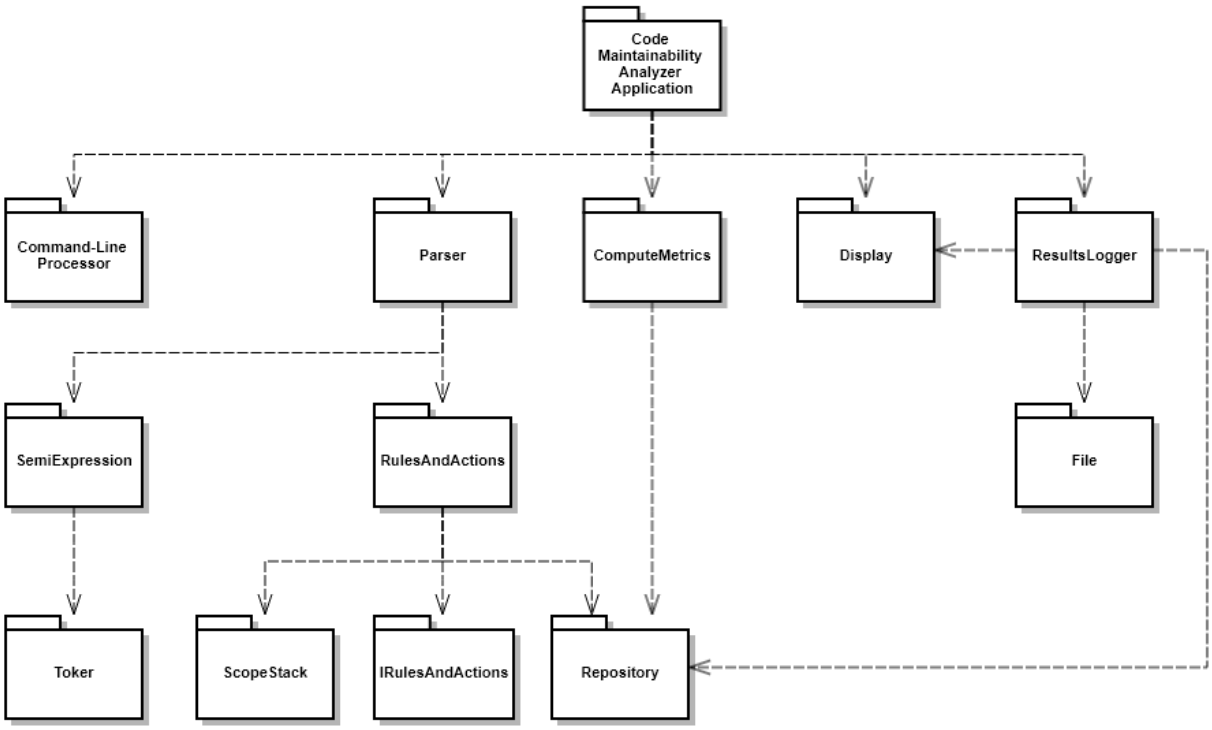

Figure 6: Package Diagram

Code Maintainability Analyzer Application Package

This is the main package of the CMA Application. This is what the end-user executes to start the CMA application.

This package is responsible for providing the ability for the end-user to provide command line arguments to the application. It's also responsible for passing those command-line arguments to the Command-Line Processor package for validation

The Code Analyzer Application Package interacts with the Command-Line Processor package to pass end-user parameters for validation and processing. It also interacts with the Parser to parse the file list provided by the Command-Line Processor. Once the files are parsed and based on the command-line arguments, the CMA application interacts with the ComputeMetrics package to perform the computation on the Repository data to compute the metrics requested by the end-user. Finally, the CMA application interacts with the ResultsLogger package to show the metric results.

Command-Line Processor Package

This package handles command-line arguments inputted from the end-user.

Description:

The Command-Line Processor Package is responsible for processing all the command line arguments provided by the end-user. This includes validating the input, providing usage text (help), and processing the arguments. Validating input includes ensuring the arguments are in the correct format and have valid data. Once validated, each argument is processed and saved so that they may be accessed by the CMA Applicaton package.

Interactions:

This package does not depend on any other package. It does, however, make available to other packages, the options desired by the end user; i.e. MI coefficients, files to parse, logging output.

SemiExpression Package

This package extracts the semi expression given a series to tokens.

Description:

The SemiExpression package formulates a semi expression which is a list of tokens from a file. Semi expression are token'ized code lines (i.e. line delimited by newline, start/end code block, or semi-colon).

Interactions:

The SemiExpression Package depends on the Toker package to return each token within the input file. The SemiExpression package is used by the RuleAndAction Package to test to determine if rules are applicable and perform actions on the rules.

Toker Package

This package is reads file-lines and returns tokens.

Description:

The Toker Package takes as input a file (or string for debug), and reads a line, and splits it based on space delimiters (space, tab, backspace) to produce the relevant token list. By "relevant", the Token Package will eat punctuation and symbols to return only the necessary information needed for making parsing decisions.

Interactions:

The Toker Package has no non-system dependencies. It does provide to tokenized data to dependent packages. Its primary consumer is the SemiExpression package.

Parser Package

This package is responsible for parsing files.

Description:

The Parser Package is responsible for executing both Pass1 and Pass2 of the code parsing. Pass 1 is focused on parsing for type / name classification and Pass 2 is focused on class relationship information.

Interactions:

The Parser package depends on the SemiExpression and RulesAndActions Packages. This package interacts with the CMA Application package; i.e. the CMA Application executes the parsing activity.

RulesAndActions Package

This package contains the parsing rules and actions.

Description:

The RulesAndActions Package is responsible for containing the rules for Namespace, Class, Function, Anonymous Scope, Public Declaration, Leaving Scope, Coupling (inheritance, aggregation, composition, use) and Cohesion.

Interactions:

The RulesAndActions package depends on IRuleAndActions which is an interface (contract) that defines how rules and actions must be implemented. RulesAndActions also depends on the ScopeStack package for code scope tracking and the Repository for state information and logging of parsed results data.

ScopeStack Package

This package maintains scope location.

Description:

The ScopeStack package provides the ability to track scope within the source code.

Interactions:

This package has no non-system dependencies.

ComputeMetrics Package

This package computes metrics required by the Code Analyzer Application.

Description:

The ComputeMetrics Package accesses the results data from the Repository to compute Lines-of-Code, Complexity, Coupling, Cohesion and Maintainability Index and then place them back into the Repository.

Interactions:

The ComputeMetrics Package depends on the Repository to acquire the results data for metric computation. Additionally, it expects to receive, from the Analyzer Application, the coefficients which the user specified for the MI calculation.

Repository Package

This package encapsulates a storage mechanism to track expressions and hold parsing results.

Description:

The Repository package is responsible for storing the current semiExpression element which is being analyzed and also storing the parsing results. Further, the Repository will be extended to allow the storage of the parsing results from File Parse Pass 1 and File Parse Pass 2 package.

Interactions:

The Repository package is not dependent on any non-system packages. It is the most interacted with package in the system since parsing and results information is stored in the Repository. It used by the RulesAndActions, ComputeMetrics, and ResultsLogger packages.

Display Package

This package outputs text to the console.

Description:

The Display Package includes functionality to display lines of text to the console.

Interactions:

This package does not have any non-system dependencies. It does, however, interact with the Code Analyzer Application to output command-line arguments and also the ResultsLogger when console output is desired.

File Package

This package outputs data to file.

Description:

Given some textual data, logs data to a file. This package creates a new file, outputs data to that file.

Interactions:

This package does not have any non-system dependencies. This package interacts with the ResultsLogger when file output is desired.

ResultsLogger Package

This package formulates the final output to be logged.

Description:

The ResultsLogger Package is responsible for creating formatted output for the code metrics results. Formatted output will be made available on the Display or to a File.

Interactions:

The ResultsLogger Package depends on the Command-Line Processor Package to know whether the end-user would like to output the results to the console or a specific file. The ResultsLogers retrieves metric results with the help of the Repository Package.

Critical Issues

Issue #1: Single Threaded Application Design

The CMA application is a single threaded design. The area where this is most evident is in the parsing activity. The application parses one file at a time. Further, since the design requires for a two-pass parser, we parse the files twice as much in a serial manner. Under small loads this performance consideration may be unnoticed. Parsing very large number of files, however, the single threaded design may not scale very well; i.e. for very large number of files it may take a significant amount of time to parse the files.

Mitigation:

At this time, this is how the application is implemented. In the future, it would be better to introduce threads to the application to handle parsing several files in the same parsing pass in parallel. For the use of this class and for small file sets, this performance drawback is a low risk. For enterprise class uses, this would be a major issue.

An alternative work-around to the issue is to spawn multiple CMA applications and partitions which files to process. Since each application will execute in its own process, this can achieve a similar result as multi-threading the application. Of course, this is barring the process creation overhead and the ability to isolate class dependencies for a give file set.

This is a low risk issue.

Issue #2: Windows Execution Platform

The CMA application requires Windows to run. While this is ok for the purposes of this class, software is often times developed on different host OS platforms such as Linux. Additionally, many continuous integration systems are non-Windows based so having a Windows only based CMA application limits its execution on non-Windows platforms.

Mitigation:

As stated, for this class, this should be a non-issue since we are only developing in Windows and scanning C# files. As an alternative the Mono project may be a viable way to run the CMA application in a non-Windows platform.

Additionally, in the future, it may be worthwhile to implement REST APIs that third-party environments can call into the CMA application and provide results back to their host OS environment.

This is a low-risk issue.

Issue #3: Identifying Non-system Dependencies

The CMA application File Parse 2 will look for class relationships. If it finds them in the source code that it parses, then it knows that this is a non-system dependency and it should take it into account when calculating its metrics; e.g. Coupling. If the dependency is in another source tree or part of another project that is not included in the scan, the dependency will be lost.

Mitigation

To get around this issue, all files that are part of the source code, including all dependencies, should be included in the scan. This can make the scanning set much larger, but without all of the dependencies the metrics may not tell the whole truth about the source being scanned.

This is a low-risk; mitigated by documentation about CMA application best practices.

Issue #4: Intention vs Metrics

The CMA application give a final Maintainability Index. This is the sum of all metrics metric. While this can be use an indicator of overall source code health, it can be misleading if there are reasons why the software was written a certain way that drives up one or many of the LOC, Complexity, Cohesion or Coupling metrics.

Mitigation:

Work with the software development team to use the "appropriate for the project" coefficients to appropriately weight each metric. This method works for small sets of source code where one set of weights can be applied to a set of source code, but it falls short when considering projects of disparate origins. In that case, running the CMA application for individual sets of code may work better. The only downside is that if there are dependencies from one source code project to another, we would lose the dependency metric data since parsing repositories are local to the CMA application.

In the future, it would be a better design to allow for a configuration file as the input and allow for coefficient specification per file sets.

This is a medium risk.

Issue #5: Future Proofing

The current Parser is a very C# parser. It recognizes and classifies types to the best its ability as it understands C# today. In the future, and perhaps even now, there are newer C# standard that should be kept up so that the parser can appropriately accommodate those code constructs into the applicable metrics. As it is right now, the

C# parser will skip code it doesn't comprehend. This may be limiting if developers take advantage of newer feature or feature not implemented by todays parser.

Mitigation

There is no quick-and-easy solution to handle this. For a production deployment, a development team would need to work on a set of rules and actions for specific C# version and test them using very language specific C# test files. As it is right now, this development team does not exist. The best mitigation to this issue is publishing documentation which can clearly explain how developer can extend the rules and actions to accommodate language features which are missing in the application.

Another option may be to open source the tool to, ideally, get more help and attention on expanding the application where the community thinks it needs to go. This can take a lot of time, effort and depending on the popularity of the application may or may not get any attention.

This is a medium-risk.

Issue #6: Reliability and Robustness

While all efforts are being made to make the CMA application reliable and robust, there is no validation team to validate and vouch for it. This application is being written for a graduate level class and every effort will be made so that this application gracefully handles errors and unknown parsing scenarios. Still, no extensive unit tests, system tests, stress tests, or penetration tests are being developed or executed.

Mitigation

None for the purposes of this course. If the CMA application goes to production, growing the development and validation teams would be required. Additionally, open sourcing the project, as previously mentioned, may give this project the attention from a development and test perspective. At this time, it this time the CMA application is recommended for student use and not recommended for production use.

This is a low-risk.

Issue #7: Extreme Input Load

If the CMA application is used to parse thousands or hundreds of thousands of files, the data in the repository will grow significantly. For each dependency, during File Parse Pass 2, the repository is searched for matching names / types. This can be costly relative to the number of input files. The result can significantly increase performance time of the application.

Mitigation

Isolate small independent (no dependency) sets and allow the CMA application to parse those serially; i.e. as mentioned with Issue #1, run the CMA application multiple times, but with smaller sets.

In the future, using faster search algorithms and structuring the repository for parallel multi-threaded access would help as well.

This is a medium-risk.

References

1. CSharp_Analyzer.zip
   http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Repository/Parser/
2. Study Guide – Operational Concept Document
   http://ecs.syr.edu/faculty/fawcett/handouts/CSE681-OnLine/Lectures/StudyGuideOCD.htm
3. Dependencies on GKGFX. Mozilla Rendering Library.
   http://ecs.syr.edu/faculty/fawcett/handouts/CSE681-OnLine/presentations/SoftwareStructureResearchExcerpts.pdf
4. Metrics-Cohesion, Rutgers University
   http://www.ece.rutgers.edu/~marsic/books/SE/instructor/slides/lec-16%20Metrics-Cohesion.ppt
5. Project #2 – Code Maintainability Analyzer
   http://ecs.syr.edu/faculty/fawcett/handouts/CSE681-OnLine/SynchronousLectures/Project2.htm