Checkstyle for Legacy Applications

Technische Universität München

Master of Science
in
Computational Science and Engineering

Chau Chin Yiu

14th March, 2008

1st Examiner: Prof. Dr. Manfred Broy
2nd Examiner: Prof. Dr. Thomas Huckle
Supervisor: Dr. Markus Pizka
Supervisor: Elmar Jürgens

***This thesis is joint work with BMW group and itestra GmbH.**

Abstract

At BMW, there are several hundreds software engineers developing and maintaining critical business information systems with a total of 85 millions lines of PL/I code and 25 years old. The maintenance cost of the systems is huge. In order to minimize the cost of maintenance, the quality of code is a key factor. Therefore, keeping a high quality of PL/I code is the issue that we have to concern.

The thesis proposes a set of rules that can maintain the quality of code. The rules restrict bad programming styles and in a program and provide useful information for the programmers. Also, a prototype quality-checking tool – temporarily called "PL/I Checkstyle" is implemented. In order to integrate the tool with the development environment of BMW, this is implemented as an Eclipse Plug-in. Apart from PL/I code analysis, the tool supports multi-language extensible architecture, which means that it not only allows to extend PL/I analyzers, but also any other programming languages. Moreover, a sophisticated clone detection algorithm is integrated with this tool.

The tool has been tested and evaluated by potential users. The feedbacks are generally positive. The potential users thought that this rules are important and useful for checking the code quality and the tool would be helpful to find bad coding style in the development process.

Chapter 1

Introduction

1.1 Background - BMW Case Scenario

In BMW Group software development division, software engineers, day by day, take isolated tests and develop IBM mainframe based commercial software projects. There are several hundreds software engineers developing and maintaining critical business information systems with a total of 85 millions lines of PL/I code and 25 years old. The total cost of the whole systems is about 3 billions US dollars. It is a well-known fact that maintenance cost is the majority of the software development process, about 80% of the total cost. It is very obvious question that might ask – How to reduce the maintenance cost? However, it is not trivial to answer. Maintenance refers a bunch of activities that is related to the tasks of "maintenance", for example, bug fixing, modifying or adding source code, enhancement of the software etc. Therefore, shortening the time of doing all these activities is one of the solutions to the problem. Then, the next question is what kind of factors can shorten the time to finish the tasks which is related maintenance activities. The issue of code quality must be one of the crucial factors that have to be concerned. Imagine the case of BMW; let us assume that there are about 500 software developers, in order to maintain 85 millions lines of code of PL/I source code, every single programmer or software developer is responsible for about 170,000 lines of code. This is not a small amount. If one A4 page contains about 40 - 45 lines, there is about 3800 – 4250 A4 pages for each software developer. What if one of them was asked to modify the code or add a new feature in their software, but the code quality is concerned as bad? (i.e. all kinds of bad practice found, like duplicated code, over nested loop, using keywords as identifier name, too many LOC* in one function or procedure). Would it be easily to let the programmer or the developer understand the logic of the code if these bad practices happen? In order to make the code become easier to be understood, all these bad practices should be identified and warn the programmers or developers to avoid doing it. Therefore, during the development process, legacy checkstyle application would be a great help for the programmers and software developers to identify the bad practices, which could cost a lot in the maintenance.

Legacy quality checking tools like Checkstyle[2], findbugs[3], PMD[4] help programmers to write software that adheres to certain coding standards. For Java, Checkstyle provides 129 Java coding rules checking, such as checking for empty blocks, illegal exception handling, and comments. It is broadly accepted that these tools contribute to increased software quality in numerous and diverse software projects.

Nevertheless, while a large fraction of the existing mission critical software systems in BMW, as mentioned above, is written and maintained in languages like PL/I, these modern quality engineering tools can only check Java or C/C++, but not PL/I. Among the reasons for this are the difficulty to analyze PL/I code, the tight integration of these tools into modern development environments (Eclipse), and the general negligence of older software systems. Therefore, before having a PL/I quality-checking tool, we have to solve these problems.

1.2 Objectives

The goal of this master thesis is to find PL/I programming style and rules that affect the maintainability of programs. Also, based on these style and rules, a quality-checking tool for analyzing PL/I programs is needed to be implemented. The tool, similar to the above mentioned tools, it should be able to read user provided configurations of coding rules, check PL/I code for based on these rules and deliver reports about possible violations. Since 2007, BMW software development division started to introduce IBM Rational Developer for System z (RDz) [4] for their PL/I Software developers, which is actually Eclipse-based development environment. This implies that the tool can be the Eclipse Plug-ins that sits on top of the open-source Eclipse development platform.

In order to achieve the goal, the followings tasks are necessary:

- Identify the PL/ coding rules
- Design an software architecture with high extensibility (more rules can be added easily)
- Implement the prototype as an Eclipse Plug-in
- Evaluate and Test the results

1.3 Structure of the Thesis

Chapter 1 – This chapter introduces the BMW case scenario and clarifies the objectives of the thesis.

Chapter 2 – This chapter presents about some related works and basic information, for example, the maintenance activities, coding standard, quality checking tools, PL/I introduction, Eclipse architecture.

Chapter 3 – This chapter is about the requirements of the PL/I quality checking tool. It would be discussed the functional requirement and the technical requirement of the tool. Also the challenges and difficulties of the development will also be discussed in here.

Chapter 4 – This chapter would discuss the PL/I rules that should be checked in PL/I program. Each of the rules would be discussed the reason why the rule is important to have and how to implement it. Some of the rules that have not been implemented would be discussed the difficulty to be implemented.

Chapter 5 – This chapter is about the architecture of the application. The architecture of the tool would be discussed in detailed. For those who want to extend the prototype, this chapter would also discuss how to extend the PL/I analyzers and even other languages analyzers like COBOL, Java.

Chapter 6 – This chapter would present the prototype of the PL\I quality checking application. The features and performance would be discussed. The details about the implementation would also be discussed.

Chapter 7 – This chapter is about evaluation of the tool. Testing result is the main discussion of this chapter.

Chapter 8 –This chapter is the conclusion of the thesis and future work is going to be discussed.

Chapter 2

Related Works

2.1 Software Maintenance

In order to know how to develop an effective and useful legacy checkstyle application, the basic idea of software maintenance and the corresponding activities should be understood. It is well-known fact that there are a lot of software development process models in the field of software engineering, like waterfall model, V-model, spiral model, top-down and bottom-up design etc. In a normal software life cycle, as shown in figure 2.1, it can be divided into 5 main tasks.

1. Requirements – specifying the requirements of the software
2. Design – designing the architecture of the software
3. Implementation – implementing or coding the software
4. Verification – testing and debugging (AKA validation)
5. Maintenance – maintaining and enhancing the software

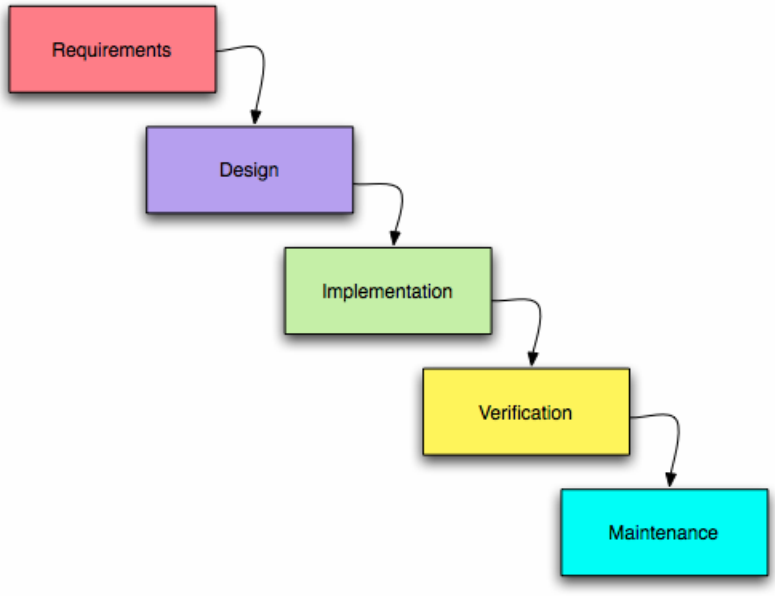


Figure 2.1.Software Life Cycle

Although there are a lot of development models, which specifically fulfill different development requirements, the cost of maintenance in various software systems is still significant share in the total cost of software development process. Software

maintenance activities typically consume 80% of the cost of a software system [5]. Perhaps, the term of “Maintenance” cannot indicate the exact and precise work or task during software development process. Apart of fixing bugs and modification of the codes, “Maintenance” in here also contains the meaning of “Enhancement of the software”. It means maintenance activity includes adding new functionalities into the existing software system. And if such maintenance activities consume excessive amounts of time and manpower, this means the maintainability of the software system is low. According to Standards of ISO 9126 [6], maintainability is the set of attributes that bear on the effort needed to make specified modifications and the set of quality attributes includes understandable, analyzable, changeable, testable, and stable, or more detailed ones such as consistent and concise naming [7].

All the quality attributes mentioned in above are most probably introduced during the implementation process, initial development or caused by long-term decay. If these defects are found in the software system, they are usually very difficult to correct the code defects. Therefore, it is necessary to continuously monitor the quality of software systems, in order to prevent the defects. Also, during the design and implementation process, quality-checking tools for developers are important for assessing the quality of software systems and control the code standard of the systems, so that it will enhance the maintainability of software systems.

2.2 Define and Measure Maintenance Activities

Since maintenance activities are dominant of a software life cycle, the economic impact of maintenance activities must be significant in software systems. However, to define the cost and economic benefit of such maintenance activities is very difficult to do. In order to identify the factor, which has the significant cost, we have to define and measure the maintenance activities. There are plenty of researches, which try to solve the problems.

The figure 2.2 shows the maintainability branch of Boehm's Software Quality Characteristics Tree [9]. This model is a mixture of the system characteristics and activities. In the figure, gray boxes refer to activities and white boxes refer to system characteristics. The model is interpreted as follows: When we maintain a system we need to modify it and this activity of modification is somehow influenced by the structure of the system. This model may give some ideas on how maintainability looks like. However, the semantics of the edges do not have a clear meaning on the relation of two nodes. And this problem is all because of the mixture of activities and characteristics.

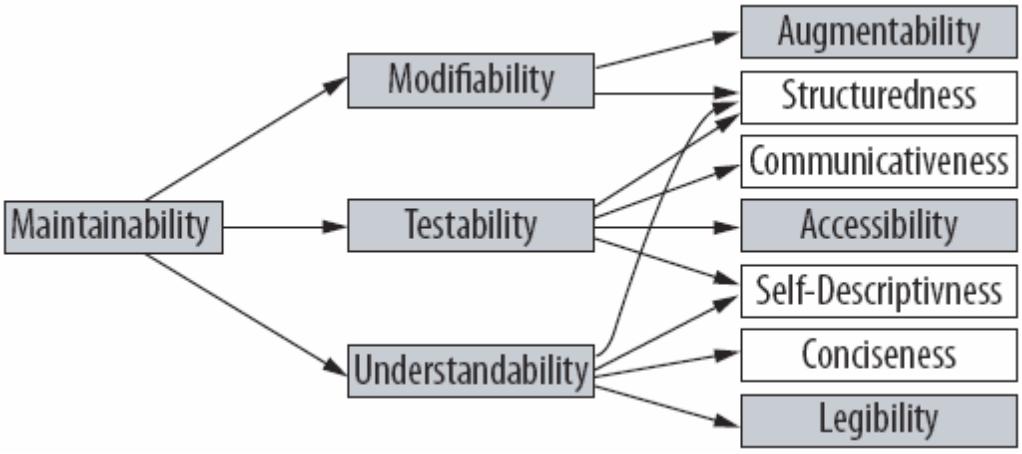


Figure 2.2 Software Quality Characteristics

F. Deissenboeck and M. Pizka introduce software product quality models which separate activities and characteristics. [8] This model can more precisely define and measure the maintainability. The key design principle is the strict separation of activities and properties of the system. This separation helps to identify quality criteria and allows to reason about their interdependencies and their effects. The application of this quality modeling approach in large scale commercial software organizations helped to effectively reveal important quality shortcomings and raised the awareness for the importance of long-term quality aspects among developers as well as managers. The following figure shows example activities and facts trees [8].

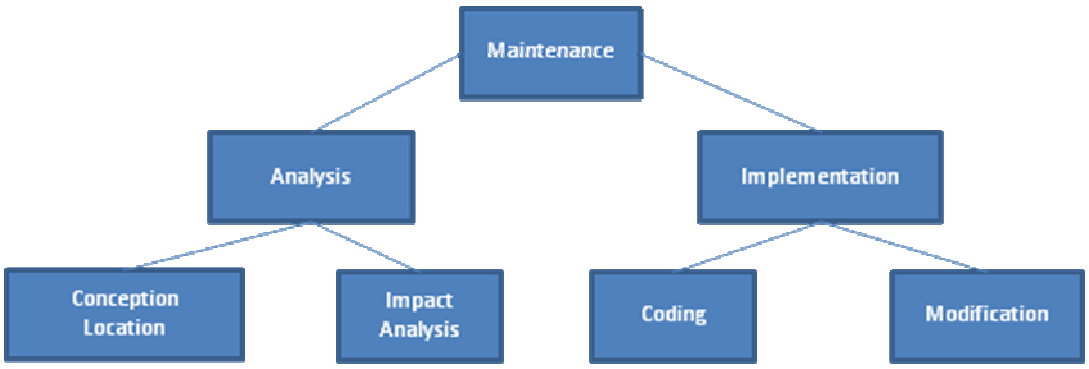


Figure 2.3a activities (up) tree

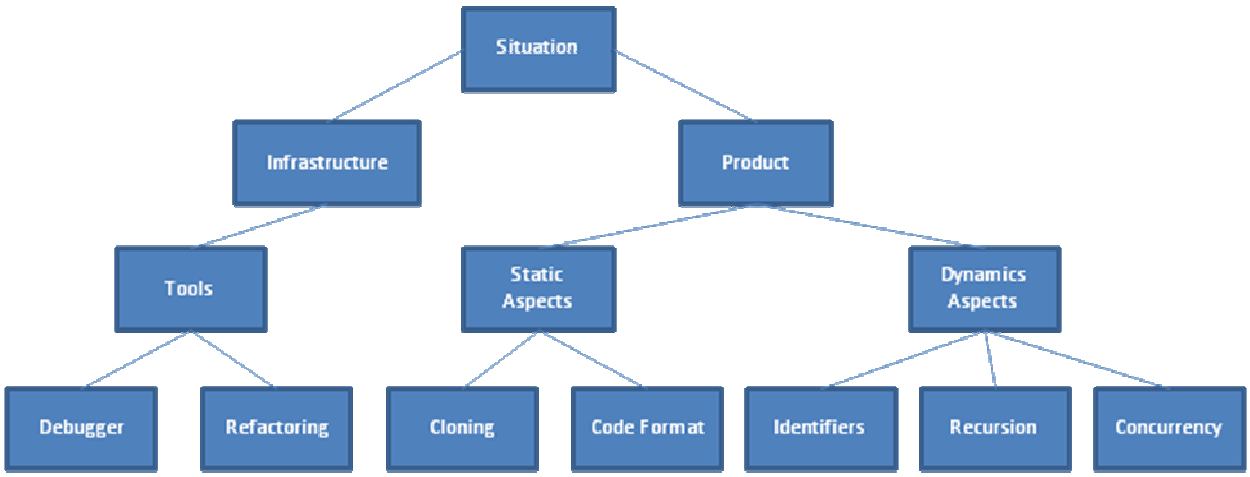


Figure 2.3b facts tree

And based on the model by Deissenboeck and Pizka, the interrelation between two trees can be expressed by a matrix as shown in figure 2.4. The matrix points out the relation between sub-fact and a specific activity and how facts affect activities (here simplified as true/false), allows to aggregate results from the lower level onto higher levels in both trees because of the unambiguous semantics of the edges, and also allows to cross-check the integrity of the model. In order to understand the figure deeply, there is an example to explain. The matrix states that tools don't affect coding (normally which is not true). It might be due to the incompleteness of the example or imply tools do not integrate with the development environments.

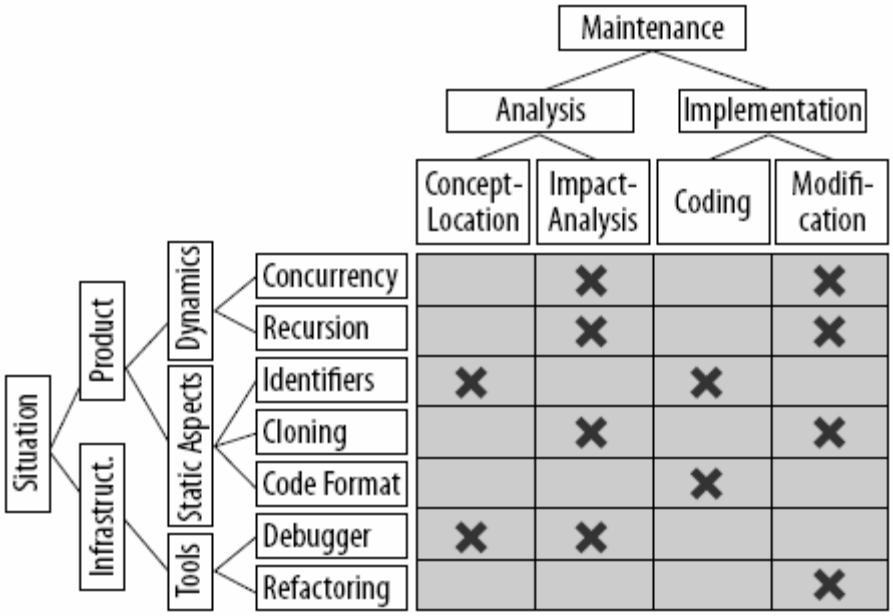


Figure 2.4 Matrix model explaining maintenance efforts

Also, at the same research group, [11] has defined a generic process model (Probabilistic Process Analysis Model) based on absorbing Markov chains that allows analyzing the economic benefit of software process variations. The model explains the conflicts between process steps and reiterations of development activities. In their research, they described that development activities are similar to tasks executed by an operation system. Due to the objectives of the overall process and limited resources, there are constraints on the order of the activities in return of the need for coordination. Every transitions from one activity to the succeeding activities are can be measured by probabilities. However, there may be a "Looping" between the activities, due to failure or incompleteness at the certain stage. As shown in figure 2.5, describes the loops (cycles) practically found in software projects. For example Developer implements some code, test it and then go back to implementing more code or fix existing code. In order to reduce the "Looping" probability, legacy checkstyle application can provide a helping hand to the developers.

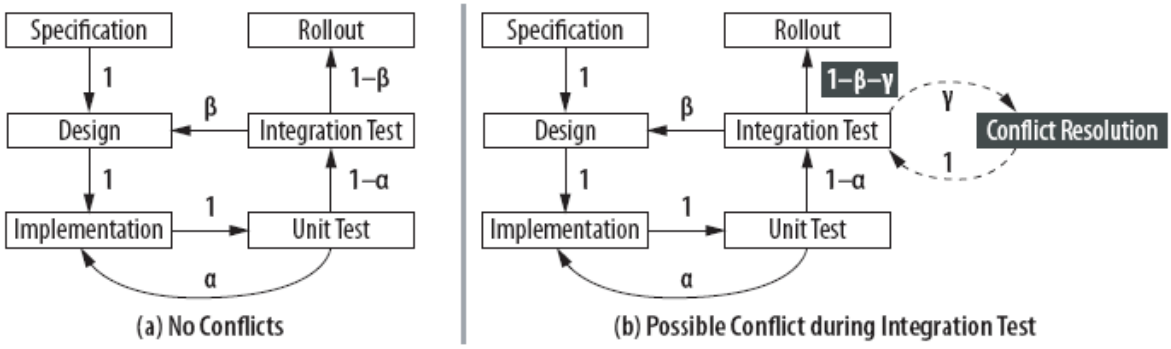


Figure 2.5. Example Processes

2.3 The Importance of Legacy Checkstyle Application

It is a well-known fact that maintenance cost is a majority of the software development process, about 80% of cost. It also means that maintenance activities are the most significant activities during the development process. Any solutions that can reduce the amount of the maintenance activities is a good solution to minimize the cost of maintenance. Therefore, Legacy checkstyle application would be considered as an effective and efficient solution to reduce the maintenance cost.

As Deissenboeck and Pizka's model mentioned above, maintainance activities can be divided into analysis and implementation. And analysis and implementation can also be divided into some sub-tasks like concept location, coding, modification etc. Every programmer or software developer, even the experienced one, needs time to understand the logic of the program, so that to finish the tasks that they were asked to do. The problem is the code quality affecting the time to let the developer understand. For example, if the code has very few comments but with many over-nested loop

blocks nes, the other developer would be hard to follow up the work on the code. Therefore, an useful legacy checkstyle application can remind the developer to avoid commiting these kind of bad programming practices or styles. Also, the application can keep the code quality in a certain high level, which can make sure that the maintenance time or cost would not be over the expectation. And for the future modification or software enhancement, developer would be easily to figure out where the code might have bigger opportunity to have error.

2.4 Programming Standards and Style Guidelines

The reasons for why programming stardards or style guildlines exists are as follows:

- As mentioned above, 80% of the lifetime cost of a piece of software goes to maintenance.
- Most of the softwares are maintained for its whole life by the different authors.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

There are so many programming standards for different programming languages available in internet. For example MIRSA [12], which is C/C++ coding standards, provide important advice to the automotive industry. Therefore, what they mostly concern includes human safety, software robustness and reliability. Also, they also suggest that system design should consider both random and systematic faults. C# coding standards by Philips Medical Systems [13] may concern the factor based on their medical needs.

All in all, different industries have different interests in coding standards, even in different companies may have their own coding convention. It depends on the needs of the specific usage. However, in general, programming guidelines is more like a programming convention than programming standards, which means that the rules of the guidelines, sometimes, cannot provide a convincing reason to follow. For example, they cannot tell the developers how much cost of can benefit by following those guidelines. Therefore, if the rules can really tell the developer why they should follow, rather than just demand those to add underscore before identifier or method based on some guru's convention, it would be much more like a measurable and valuable "standards".

2.5 Code Quality Checking Products

There is a lot of code quality checking applications, which are available to download from the Internet. For example, PMD, findbugs, Checkstyle and Lint etc, are the name of a bunch of quality checking tools, which are free to download. Most of these tools are mainly for Java and some of them for C/C++. For Java checking tools, it is much easier to implement the analyzer because of running on virtual machine and the simple grammar of the language. And its simple object model idea which has fewer low-level facilities make the program code much easier to be analyzed. Therefore, nowadays, the code quality checking tools are mostly likely to focus on Java.

The following products are the popular code quality checking tools:

2.5.1 CheckStyle

Checkstyle is a code quality-checking tool or development tool to help programmers write Java code. It can check many aspect of a source code, like the code layout issues. Now, they have already added more features on class design problems, duplicate code, or bug patterns like double-checked locking. This makes it ideal for projects that want to enforce a coding standard. However, some of the rules implementation is not sophisticated enough, like duplicate code checking, it does not support fuzzy matches, which is a strict-matches check. Checkstyle is highly configurable and extendable, which means that the users can create their own analyzers, or modify the checking standards. And Checkstyle has already been integrated with most of the IDE or Java build tool (i.e. plug-in), like Eclipse, NetBeans and Borland JBuilder etc.

2.5.2 PMD

PMD is another tool for checking Java source code. Comparing to Checkstyle, PMD is intended to find the potential problems in the code, like possible bugs- empty try/catch/finally/switch statements, and dead code etc. It also can be extended and configured by users. One thing which is more advanced than Checkstyle is JSP support, which means it can distinguish a piece of code that is a mixture of XML tags and Java code. There are PMD plug-in available for some IDE.

2.5.3 FindBugs

FindBugs is a product mainly developed by University of Maryland. This is a static analysis tool to inspect Java bytecode for occurrences of bug patterns. Static analysis means that FindBugs can find bugs by simply inspecting a program code: executing the program is not necessary. FindBugs works by analyzing Java bytecode (compiled class files), so program source code is not needed. Because its analysis is sometimes imprecise, FindBugs can report false warnings, which are warnings that do not

indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%.

2.5.4. ConQAT

ConQAT is developed by the Competence Center Software Maintenance of the Software & Systems Engineering Group in the Munich University of Technology. ConQAT (Continuous Quality Assessment Toolkit) is a toolkit for the continuous controlling of software quality. It enables the integration of various quality assessment tools or methods in a flexible way. ConQAT is not limited to the analysis of source code because the user can extend the analysis modules very easily. Also, the visualization mechanisms and architecture analysis enable both developers and project managers to track key quality aspects of software projects in an efficient and timely manner. However, it still has no integration with any IDE.

All in all, there are many code analysis tools available for different language developers to download and use. However, for PL/I, it is the exceptional case that no open source product or commercial product is found for PL/I code analysis. This is because the research of this thesis is a new topic in the field of code analysis.

2.6 PL/I Introduction

In this section, we will have a brief introduction of PL/I and discuss the reasons why there is no PL/I code checking tools available and the difficulties of implementing the code analysis software application.

PL/I ("Programming Language One") is an imperative computer programming language, like C, C++, Java. It is designed for scientific, engineering, and business applications. It was developed by IBM since 1964. It is definitely an old programming language. PL/I is mostly used on mainframes computer. There are still other versions for DOS, AIX and Unix etc. It is one of the most feature-rich programming languages. It is a block structure language, which means each of the programs is constructed by a series of blocks and every statement is embraced by a block. Also, the principal domain is data processing. It supports recursion and structured programming. The language syntax is English-like and similar to FORTRAN and COBOL. It is suited for describing complicated data formats, with a wide set of build-in functions available to use. Despite of the powerful features in PL/I, nowadays, there is not many commercial or academic organization using it. It is mostly used in middle Europe and some traditional industries, like BMW.

2.6.1 Difficulties of analysis of PL/I code

Before we are taking about the difficulties to analyze the code, we have to know that the basic idea of analysis programming code.

To analyze a program code, as shown in figure 2.6, there are 3 components needed – Lexer (scanner), Parsers and Analyzer. Firstly, lexer is used to scan all the tokens from source code files. It needs to distinguish between syntax and non-syntax characters or strings. Secondly, the output tokens stream go to a parser, which recognize the structures of this tokens stream. It can also generate some call graph, abstract syntax trees etc. finally, the output of parser would go to the analyzer, which do really analyze the parser results and get useful information for generating the output for the programmer.

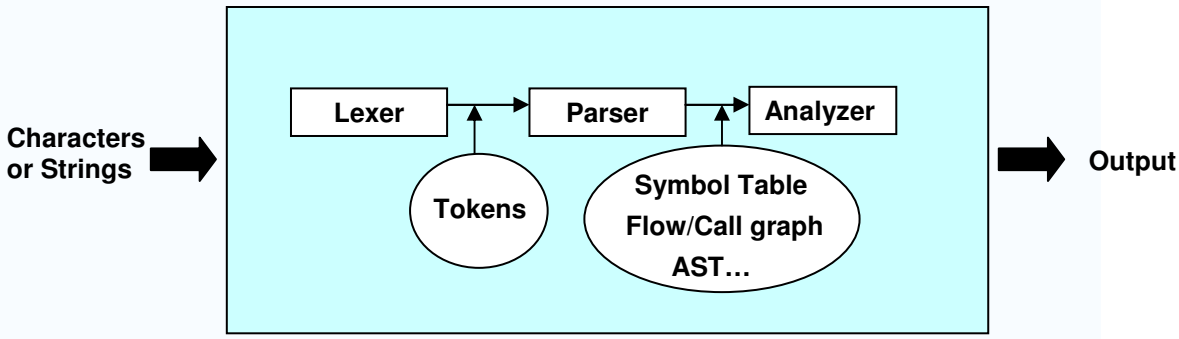


Figure 2.6. Code Analysis Principle

In the case of PL/I, to implement a parser, it is difficult and time-consuming. It is because of following reasons:

- The compiler cannot tell if a statement is a declaration or an executable statement until the ending semicolon is found. However, it could be several lines later.
- The PL/I keywords are not reserved. It means that programmers could use them as variable or procedure names in programs. Therefore, each time, the compiler finds a keyword; it had to determine if it is being used as a keyword or as a name. Therefore, for code analysis, unless programming a complete compiler, it is impossible to check if the keyword is used as keywords itself or just a variable name by just scanning the source code file.
- The sophisticated macro facility can even change the source code during the compile time. Therefore, it is impossible to ensure that status of code.

A PL/I compiler is typically two to four times as large as FORTRAN or COBOL compilers, and also that much slower. Therefore, it is not a good idea to implement another compiler to analyze the code, and that's the reason why there is no solution for analysis of PL/I.

2.6.2 Development of PL/I Coding Rules, Style and Convention

Because of the difficulties to check PL/I code quality, PL/I programmers has to find the ways to keep their codes “healthy”. Therefore, some of the PL/I programmers or software managers developed a set of rules that restrict or encourage PL/I developer to follow; so that the maintenance cost of PL/I systems can be minimized. To come up with all these rules, different managing people propose different rules or style depending on their interests, which includes computational time, performance, and readability or understandability of code etc. For example, regarding to the performance reason, it is better to use IF-(nesting) ELSE IF statement rather than SELECT WHEN statement. However, in this case, from the software maintenance point of view, if there are too many cases that need to switch , it is better to use SELECT WHEN statement. [15, 16]. Another example, like GOTO statements, in some situations, programmers are asked to avoid using GOTO statement, but some situations, it can be compromised. However, from the software quality point of view, GOTO is not allowed in all situations. For PL/I guru, they have their own requirement on comments, like the date of modification of the code and which lines are changed etc. This is also one of the convention that PL/I programmers have, in order to minimize the time to maintain the code. In Contrast, software manager only requires the proper ratio of comments in the code. Therefore, the rules or conventions developed by programmer are much more preferred to performance and readability. All these rules may not have logical reasons to convince the programmer, but just a convention. On the other hand, for software manager, they are much more to the cost of development; therefore, they consider the rules mostly related to software quality. Of course, for the both parties, they have common interests that they really consider those rules or style is worth to follow, like the nested looping group should be larger than certain extent (e.g. < 5 loop level). On the other hand, development of rules is somehow dependant on the technology of the development environment. Some of the rules may be redundant when a latest version of IDE is released. For example, Rational Developer for System z (RDz) is an IDE, which is based on the Eclipse platform. Therefore, they have a lot of functionalities that help developer to stick with certain rules. For example, the margin fixing and the local syntax check which helps a lot to restrict programmers to program in a certain rules or style.

In short, in order to develop the rules of programming PL/I, the point of view of both programmers and software managers should be taken into account. However, for those rules or styles that exists because of conventions or traditional manner, should no longer have valuable to follow. Once the rules or style cannot prove their economical value, what the point to follow it? When we are taking about regulation, rules or standards, a rational, objective and logical explanation is needed. Convention and traditional manner should not be considered as a reason to explain why the rules should exist.

2.7 Eclipse Plug-in Architecture

As mentioned in the previous chapter, the thesis research is a joint work in BMW. And they started to change their development environment into RDz, which is based on Eclipse. Therefore, in order to integrate the solution into Eclipse, this section will introduce the Eclipse plug-in architecture.

The Eclipse Platform is a development framework. This framework provides an open source platform for creating an extensible integrated development environment (IDE). This platform is well-designed and highly extensible architecture. This platform allows any people to build their own tools which integrate with the environment and other plug-in tools. This architecture is very creative and flexible. Anyone who wants to make their tools to integrate with Eclipse platform is just need to define their tools in certain extension point, and this mechanism is all done by a XML file [18, 19]. Therefore, it is very easy to get familiar with the extension mechanism. Also, this is the reason for which makes Eclipse growing from time to time. Nowadays, there are up to 1034 plug-ins available to download in Eclipse Plugin Central website [20].
The extraordinary idea of integration of tools with Eclipse is the plug-in. Apart from small run-time kernel, everything in Eclipse is a plug-in. This means that a plug-in you develop integrates with in Eclipse as the other plug-ins. This makes all features are created equal.

However, the workbench and the workspace are two indispensable plug-ins of the Eclipse platform. It is because they provide the critical extension points that are mostly used by the other plug-ins. The following figure is a basic idea of how the plug-in that we make co-operate with Eclipse.

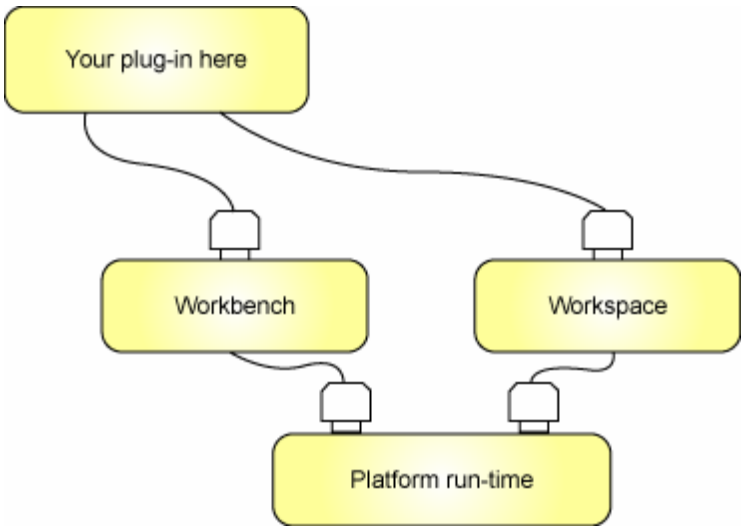


Figure 2.7. The Eclipse Workbench and Workspace: essential plug-in support

The Workbench and the Workspace not only provide Eclipse components that can be extended by other plug-ins, but also there is a debug component that will let your plug-in launch a program, interact with the running program, and handle errors. There are all about to build a debugger.

In the case of the solution provided in this thesis, there will be a detailed discussion of how integrate the solutions with the Eclipse platform in the coming chapters.

Chapter 3

Requirements of the Solution

In this chapter, we are going to talk about the requirements of the solution and the challenges of implementation of the solution. The requirements can be divided into two parts; one is ***Functional Requirements***; the other is ***Technical Requirements***.

3.1 Functional Requirements

In order to develop a legacy checkstyle application, there are two main parts of the functional requirements – establish effective rules and graphical user interface.

3.1.1 Establish Effective Rules

For the legacy checkstyle application, the rules being checked are very important. However, in some other legacy checkstyle application, there always happens that the applications provides a lot of "boring" checks for the user. Sometimes, it happens warning message "flooding". This is not a good idea to have thousands rules to check in the application. It does not make sense to have 10 times more rules than the rules of language grammar. It does not help the programmer, but annoy them. Of course, if too few rules to check, it cannot provide a good quality analysis for developers. Therefore, to judge or to evaluate whether a rule should exist or not, we have to consider the following issues:

1. Economic Value
2. Technical Feasibility
3. Manageable, Measurable and Quantifiable
4. Software Quality related
5. Representative or Redundant

Before establishing a rule, asking a critical question is needed. "How much time or money can it be saved, if take this rule?" .Regarding to economic value, this question is needed to ask when decide a rule if it is valuable. Valuable rule means that it really costs time and money. This also means the rules with economic impact. Any programming behaviours or style cost unreasonable manpower or time should be avoided. Therefore, the rules that can identify those problems are worth to have. On the other hands, any behaviours or styles which reduce time and money are worth to have.

Technical Feasibility, obviously, is about whether the rule can be implemented to check. However, it is not necessary meaning the difficulty of the implementing the analyzer. It is also about whether the rule can be regulated. For example, we can encourage the programmer to give a reasonable or understandable procedure name or variable name in their source code. Unfortunately, we cannot restrict them to use some non-sense names or tell them if the naming is non-sense or not. This is a lot more than a computational algorithm can do. Therefore, in this example, it is not feasible to implement.

Asking if the rule is manageable, measurable and quantifiable is also needed. It is important that the rule can tell some numbers, and the numbers can help you to fix the problem or improve the software quality. For example, number lines of code within a procedure, this number give the user a quantity that can justify if the code is bad or good, so that the user can modify the code. However, there are some quality checking applications, which give some rules, like "Quality grading should be at least 5 out of 10" or "Health index should be over 60" and then, the result would be a sort of marks or grading. From a software manager point of view, it may give information to judge if the software system quality is good or not. However, for a developer, the rule result cannot tell them where should be modified or how to manage in order to have higher grading. Therefore, a rule that can tell the numbers and can be easily identified the problem by programmers is very crucial.

A rule that is related to software quality factor is also needed. According to some research papers [21, 22, 23], Software quality factors of a source code or software system includes understandability, conciseness, maintainability, reliability, testability, usability, structure, efficiency and security etc. If a rule can enhance the software quality factors, the rule is worth to keep. For instance, consistent indentation can help understandability of source code; it is a good idea to check if the indentation is consistent.

Finally, some rules are more or less duplicated; however, it is not easy to justify. For example, comment ratio and numbers of lines of comments are actually telling the sample information. However, comment ratio is more representative than numbers of lines of comment. It is because statement number should be directly proportional to comment (i.e. more code; more comments). Justifying whether the comments is sufficient, it needs to compare to the amount of source code. Therefore, seeking for a representative rule but not redundant one, make simple and accurate to the developers and programmers.

3.1.2 Graphical User Interface

Another functional requirement is Graphical User Interface (GUI). GUI serves the interaction between the users and core application. It enhances the efficiency and ease of use for the underlying logical design of a stored program. A well-designed

application should be functionally independent from and indirectly linked to core application functionality, so that GUI can be easily customized. This design pattern is called Model-view-controller. For a plug-in legacy checkstyle application, Model-view-controller design model is still applicable. This will be discussed in details in the next section.

In this case, user triggers off the code analysis or checking. The core analyzers do a course of calculation and generate a result data model. Then, this data model is display on a view. This is a rather typical work flow of a software application. Nevertheless, to develop the part of GUI is not that easy to fulfill the requirements of well-designed, user-friendly, helpful and useful. It happens very often that the functionalities in the GUI part are more than enough, which means it is too complicated for the users. Although it is not strong evidence to prove, it is very interesting parable to show the redundancy of GUI development. 80% of functionality only used in 20% of daily work; and 20% of functionality is covered over 80% of work, by the Pareto principle. In order to minimize the redundancy of the GUI, “simple”, “easy” and “clear” are the buzzwords in development of graphical interface.

For GUI part of legacy checkstyle application, it mainly has 3 components. First is the trigger; the second is the configuration; final is the output display. For the trigger part, it is better than to keep it simple, like only one click or one button to start the analysis. For the configuration part, in this case, it would be the preference page and the configuration file, which can configure the analysis tools. In the preference page, it is better to make it extensible, like a scrollable page. It is because the number of tools will be getting more and more. If it is a fixed preference page, it would be not fulfill the requirement of extensibility of the whole application. Also, the instruction should be clear, so that user will not be confused. In the case of output display, it is rather complicated. It depends on the type of data. And it should interact with the text editor, so that make the development process much more smooth. Also, in order to minimize the side effect of using the application during the development, high resolution graphical process should be avoid, because it would consume memory which may cause sluggish movement of graphical object on the IDE.

3.2 Technical Requirements

Technical requirement are concerned about the approach of the design, the application architecture design, and implementation of the design.

3.2.1 The approach of the design – Eclipse Plug-in

It is mentioned in the previous chapter, since this is the research project in BMW, which start to change their PL/I development environment into Rational Developer for System z (RDz) based on Eclipse platform, it is a good idea to integrate the

solution into their development environment. This means Eclipse plug-in would be the best choice to approach the solution.

On the other hand, the reason why Eclipse plug-in would be the best choice of the starting point of the solution is due to two key points:

1. Eclipse has dominant market share. According to the BZ Research survey [24], from 2003 to 2005, Eclipse was getting significant increment of market share, from 34.5% to 65.1 %. Although this is the research of Java development environment study, since the Eclipse platform can allow different language development environment plug-in, the figure is still creditable and referable for the other language development environment.

2. Eclipse plug-in features can make the application expandable. Also, because everything is plug-in in Eclipse, the development environment would be getting complicated, like multiple language development projects might happen. For example, a web services, using Java, C/C++ to develop back-end and HTML, JSP, JavaScript or Actionscript for the front-end. If the application is started with Eclipse plug-in, it is much easier to interact with the other plug-in tools. It is good for the future development.

All in all, either for the need of BMW or for the future development, Eclipse plug-in approach would be a good idea to start with. Also, for the development issue, Eclipse platform provides its flexible and expansible architecture which largely ease the difficulty of implementation of the application.

3.2.2 Application Architecture Design

In this section, it would mainly discuss the requirement of the architecture design, but the actual architecture of the application which will be discussed in chapter 5 in details. There are three things that the design should be taken into account.

1. Model-View-Controller architecture
2. Extensibility of the analysis tools
3. Multi-language analysis support(optional)

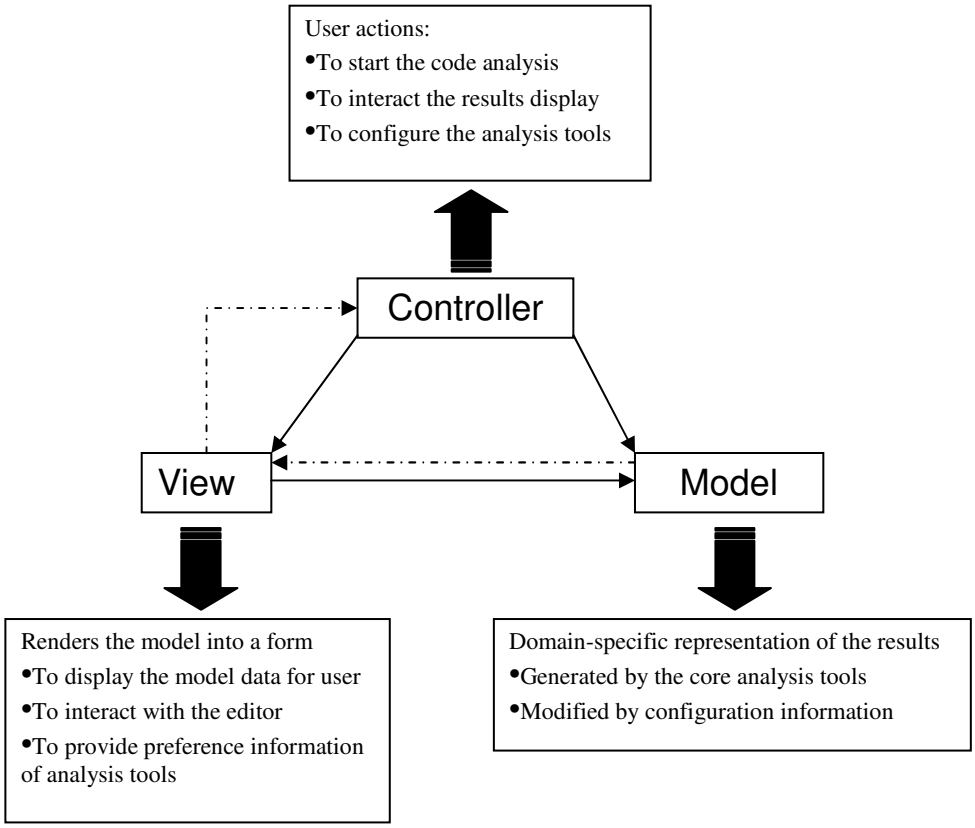


Figure 3.1 Model-View-Controller architecture

(Note: the solid lines indicate a direct association, and the dashed lines indicate an indirect association)

Model-View-Controller (MVC) architecture solves the problem by decoupling data access and the core logic from presentation layer and user interaction, by introducing an intermediate controller. In order to have a robust architecture that benefits in the future enhancement, MVC is needed in the application.

In the above figure, show the relation between model, view and controller and what exactly need to do in the application. For the model, this is domain – specific representation of the results that is generated by the analysis tools. This data model is generated by different analysis tools. And this data model will be read by the view which renders the data model into a form that is understandable for users. For the controller, it is control of display on the view and modification of the model. And it also represents the user action in this case.

Extensibility of the analysis tools

Because of unknown number of analyzer, in order to be more flexible, it is better to have an architecture that is easily to add or remove analyzers. This requirement is concerning about the future enhancement of this tool. Since the factors of software quality are diverse and it may change from time to time, having a flexible architecture will never be limited by the time and technology change. Also, no tool can fully support the all aspects of problems, so it must provide an extension mechanism that

allows users to develop their own analysis to fulfill their purpose and target need. The tool must also allow combining different analysis. For different projects, the objectives of quality assessment are different. Therefore, the tool must have a flexible configuration mechanism.

Multi-language analysis support (optional)

Although this is a project of quality checking application for PL/I, it is a creative idea to have in a quality checking tool. In the internet, as mentioned in the previous chapter, like Checkstyle, PMD, findbugs, these tools are mostly for Java. Some of them with the other programming language; however, it is not in the same application package. User has to download the application package, so that to check the specific programming language that they need. As we all know that, Eclipse has many programming environment plug-ins, and it is getting dominant in the market. Also, multi-language project is getting usual in these days. For example, a computational performance demanding project which needs fast calculating time and needs complicated user interface, is probably needed to have two languages to fulfil that requirement, e.g. C/C++ for the core algorithm and Java for the interface etc. Therefore, multi-language analysis support would be much more flexible and fulfil the requirement of different project.

Chapter 4

PL/I Quality Checklist

This chapter mainly discusses the list that should be checked or be aware. First of all, we have to define what the PL/I quality checklist is. The PL/I quality checklist is the list of criteria (rule) or information that is directly or indirectly related with code quality factor of PL/I source code. The list can provide information for the programmers that what should be paid attention at.

Before having the rules to control the quality of code, we have to identify the information and criteria that would affect the quality of code. To identify the information and criteria that is important in quality checking, we have to identify what kind of information is interesting for software manager and programmer. Time, money and manpower must be the crucial factors in a software development process. Therefore, when we consider criteria or information if this is worth to check, these three factors have to be concerned.

Also, especially for PL/I feature, the criteria and information should be specific in PL/I. Although code quality is programming language independent, which means it does not have direct linking with the design of the language, it still needs to focus on feature of PL/I. It is because, for a legacy checkstyle application, the main user is the PL/I programmers. What they care is all about PL/I programming grammar. It is necessary to pay attention at their programming convention or some small and detailed programming style. It is nothing to do with the code quality but from user-friendly point of view, closed to what they need is important.

In this chapter, we will discuss each of the rules and information in details, including the reason why they should exist, how to implement it and the difficulty to implement it. In the Table 1, this is an overview of the list of rules or information that we are going to discuss in the following.

List of rules	Description	Implement
LOC	Lines of code	Yes
SLOC	Statement lines of code	Yes
Keyword named Identifier	PL/I keywords not reserved, Identifier can be a keyword	Yes
Procedure LOC	Lines of code in a procedure	Yes
Comment Ratio	The ratio of comment to source code	Yes
GOTO statement	Usage of GOTO is not recommended	Yes
Code in comment	A piece of code which are commented	Yes
Dead Code/ Redundant Code	Non-used code	No
LOC of Dead Code/ Redundant Code	Lines of code of dead code or redundant code reflect the usability of code	No
Cloning Code	The cloning and duplicate code is not a good practice	Yes
Nested Loop Depth	The complexity of loop mechanism	Yes
Nested Loop Depth across procedure	The complexity of loop mechanism across procedure(i.e. a looping procedure call another looping procedure)	No
OTHERWISE case in SELECT statement	OTHERWISE (Default) case should exist in SELECT statement	Yes
LOC within WHEN case in S ELECT statement	Lines of code in each case (WHEN)	Yes
Code margin	Coding region should be bounded between 2 and 72 column	Yes
LOC in ON unit	Lines of Code in On-unit	Yes
Number of ON-unit and their locations	The more ON-unit ; the more complexity of code	Yes
Number of parameter in procedure	Number of parameter passing into a procedure	Yes
Stopping criteria in recursive procedure	If there is no stopping condition stated, it may happen dead looping	No
Foul Language Checking	Check if there is foul language in source code	Yes
Block/Group in Preprocessor	Block of statements in preprocessor is not good practice	Yes
Number of Block/Group	Number of block in a source file	Yes
Block-Group Depth	The Depth of a block and group	Yes
LOC in Block-Group	Lines of code in a block	Yes
Failed Block-Group	A block without END or more END is regarded as a failed block	Yes
Block/Group Indentation Consistence	Indentation should be consistent for the block	No
Calling Relation between Procedures	To see the relation between the procedures	Yes
Number of being called of a procedure	To see how many access to a procedure	Yes
Number of external procedures	To see how many external procedures are imported in source code	Yes
Loop Depth of CALL	To see the loop depth level of a call statement or a function call	Yes
Empty Block	Empty block does not make sense to exist	Yes
IF Block	Identify the block of IF-THEN-ELSE	Yes
Bit Operation in IF expression	The complexity of '&' , '|', '^' in one expression	Yes
Comparison Operation in IF expression	The complexity of <,^<,<=,=,^=,>=,>,^> in one expression	Yes
Depth of IF statement (nested IF statement)	The complexity of condition mechanism	Yes
IF statement Ratio	The ratio of number of IF statement to SLOC	Yes
Number of ELSE IF in a group	The number of ELSE-IF in a group after IF statement	Yes
Blank line ratio	The ratio of the number of blank lines to the number LOC	Yes
One statement per line checking	To see if there are more than 1 statement in one line	Yes
SELECT expression check	Data type of expression consistence checking	No

Table 4.1 Overview of PL/I Quality Checking List

4.1 Rules Discussion

In this section, we are going to discuss the rules that will be implemented in the checkstyle application. In each rule, the reasons of having the rules and the implementation methods will be discussed. The rule that has been implemented will be notified by (*) at the title of the rules. If the rule has not been implemented, the technical difficulties or challenges will be discussed in an addition paragraph with a title of "Difficulties of implementation". Also, the detailed information about LpexParser, BlockAnalyzer and IF-ELSE Block parser are attached to the appendix.

***Lines of Code (LOC)**

LOC is the lines of code in a source code file. It implies to the size of system, architecture of the software system and the design of the system. For example, the average LOC of a software application is 100. If there is a source file with over 1000 LOC, it may have some certain problems, like too much unused code, over-commented or design problems. Also, in the maintenance point of view, too many LOC is relatively difficult to read and understand. It would cost a lot maintenance cost. This information can give programmers an idea that the source file should be divided into two or more source files with an appropriate LOC.

Implementation

There are two ways to implement it. On one hand, it can be done by Lpex editor, which is built-in editor RDz. It provides a functionality which can detect the LOC of a source file. On the other hand, by using the PL/I token scanner, it can get the location of the last token in a source file. Therefore, LOC can be known.

***Statement Lines of Code (SLOC)**

SLOC is number of statement lines of code. It only counts the lines of statements, which do exactly the calculation or computation. This information can provide the exact size of the computation part of a source file. Different from LOC, SLOC ignores empty line, multi-lines statement and comments. It counts every statement which does computation. Therefore, the size of computational algorithm can be known. It is also critical information to tell programmers whether the design of a program code should be divided into several parts or not.

Implementation

To implement the check of SLOC, identifying the end of a statement is needed. In PL/I code, every statement should be end with semicolon ";". Therefore, it can simply count the number of semicolon. However, for those semicolons in comments or strings, it will be ignored. It can be easily done by the PL/I token scanner. By extracting the tokens and checking them one by one, the number of semicolon can be counted.

***Keywords named identifier**

Since PL/I do not reserve keywords, programmers can use the keywords as their variable name. This feature may make advantage on naming flexibility. However, it causes a lot of problems. First of all, it may cause some potential bugs in the code. It is duplicated with keywords. It may cause some program during the compilation time. Secondly, it makes the code difficult to understand. It is easily to imagine that if an identifier is same as the keywords, it would more or less confuse the programmers who maintain or design it. It may cause more cost to maintain a code with this kind of identifier than the one without the strange named identifier. Finally, an identifier having the same name as a built-in function or procedure can cause a lot of problems when this identifier is exported as a public variable. This will confuse the programmers who take part in the same software project. It may cause a disaster during the software development.

Implementation

To implement this checking, we have to know what the PL/I keywords are. And then put all these keywords into an intermediate storage. In order to check whether the identifier is keywords or not, we just need to pay attention at the identifier declaration, which means how to define an identifier or a variable. In PL/I grammar, every identifier and variable have to be defined by a DECLARE statement. Figure 4.1 explains DECLARE statement structure.

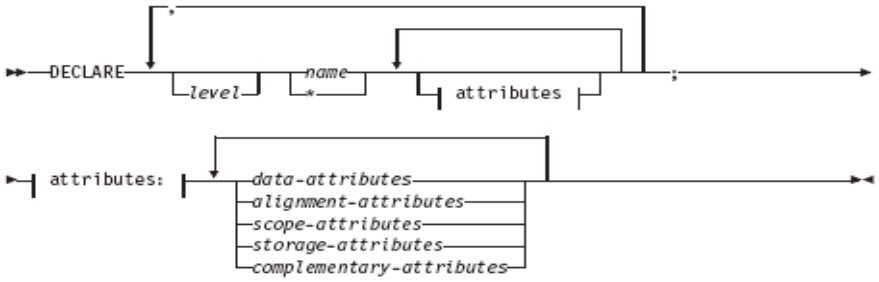


Figure 4.1. Abbreviation of DCL

There are three elements in each DECLARE statement. They are level, name and attributes. The level is a nonzero integer. If the level number is not specified, level 1 is the default for element and array variable. Normally, it is not usual to specify the level number. And the attribute specify a couple of information, as shown in figure 4.1. In this case, we just need to know whether the name is keywords or not, so the level and attributes can be ignored. Attribute can be easily distinguished in a DECLARE statement, because it is after the name. There are exceptional cases - the level and the factored name. The level is not necessary to have in the statement. Therefore, if there is an integer token right after DECLARE, this token can be ignore; then take the token after the level token. However, for the factored name, it is more complicated. For example, `declare (A, B, C, D) binary fixed (31);` A,

B, C and D are the identifiers, but inside a bracket separated by comma. If there is a bracket after DECLARE, the tokens inside the bracket will be extracted except the comma.

***Procedure LOC**

This is an indicator to show the lines of code in a procedure or function. Although it is hard to judge that over a certain lines of code is good or bad, it is obvious that a procedure or function with too many LOC is not a good practice in programming. If a procedure is over 500 lines which is almost the length of the whole source file, it will consume more time to understand the code logic. A proper length of procedure can also tell if the program code has good encapsulation. Therefore, Procedure LOC can help programmer to know whether their procedure is too large. If so, this is a good reason to restructure or re-design their code.

Implementation

In order to identify procedure block in a source code, a light weight parser is needed. However, for saving more time of implementation, LpexParser has a function – `matchToken` which can find the END token of a procedure. By scanning the tokens in source file, for the token of PROCEDURE or PROC, they will be put into the `matchToken` function and then, the output will be the END token. After finding a pair of tokens PROCEDURE – END, the LOC between PROCEDURE and END will be found.

Although LpexParser is not a complete parser, it still has some useful function that can save more time in implementation of rules checking.

***Comment Ratio**

Comment ratio is the ratio of the number of comments lines to the number of lines of code. The amount of comments in a source file cannot tell if the comment is sufficient or not. However, by comparing the amount of LOC, it is much more accurate to tell if the comment is enough or not. It is because the more codes are supposed to have more comments. Therefore the amount of comments should be proportional to the amount of code.

Implementation

It is relative simple to implement this rule. By counting the number of comments lines divided by the LOC of source file, the comment ratio of the source file can be calculated. In this case, PL/I tokens scanner is used to count the comment token. Each comment is considered as one comment. It is possible to have multiple lines comment. Therefore, it needs to know the number of LOC that the comment token has. It can simply be calculated by comparing the line number of the comment token and the line number of the token right after the comment token.

***GOTO statement**

The use of GOTO statement has been controversial issue in computer programming field. Some people claims that using GOTO statement results in unreadable and generally not maintainable "spaghetti code". Many computer scientists came to the conclusion that programs should be much more structured flow – control rather than using GOTO that “jump” to another statement.

The most famous criticism of GOTO is by E.Dijkstra [25]. He argues that unrestricted GOTO statement should be existed in higher-level languages because they make the task of analyzing and verifying the correctness of program much more complicated, especially nested with loop. Therefore, in this Checkstyle application, GOTO statement will be checked and does not recommend using it.

Implementation

It is very simple by using PL/I token scanner to scan each token from a source file. Each of GOTO token can be located and give the warning message to programmer that GOTO statement should not be used.

***Code in Comment**

Commented code is not a good practice in software maintenance point of view. For the commenting code, there are two possibilities. On one hand, the commented code is useless. It can be deleted, but the programmer forgets to do so. It is not a good practice, because it wastes a lot of space in a source file. It might confuse the programmers to understand the code. This is because comment is supposed to explain the code, but the commented code cannot provide any information for the programmer. On the other hand, the commented code is useful, for example used for debugging if it is uncommented. If it is deleted, maybe it will cause some problems when the compilation or execution configuration is changed. Therefore, it is better to inform the programmer that the code is commented. Also, for the useful commented code, it is better to change them into uncommented and control the execution procedure by using some other logic but not just modifying the source code. Moreover, for the unused code, just simply delete it.

Implementation

Firstly, Commented code has to be located by using PL/I token scanner. Each comment would be considered as one token (/*----*/). And then, by loop through in the text of this comment token to find semicolon, which normally indicates an end of statement. Therefore, simply counting the number of semicolons in one comment can get the number of statements which is commented.

Dead Code/ Redundant Code

Dead code means unnecessary, inoperative code that can be removed. It is a well-known fact that a program quality can be improved by removing dead code. It will help in maintenance by decreasing the maintained code size, making it easier to understand the program and avoid introducing bugs.

Difficulties of implementation

To detect dead code, it is relatively difficult in PL/I. It is hard to know the code if it is dead until it has been compiled. Therefore, it is hard to detect by simply scanning and analyzing the tokens in a source file. If create a PL/I compiler, it will cost too much to implement this rules. Therefore, it can only be done by detecting local dead procedure. The procedures that are not called by the other procedures in the same folder can be found. However, this method is rather limited. It is because the PL/I programmers in BMW do not have the whole set of PL/I files (i.e. the whole software system) in their local computer. They have to run and compile in the mainframe computer. Therefore, it may happen that their program may be called by the other programs that are not in their local workstation. In this case, we can just detect the procedures that are not called by local programs.

LOC of Dead Code/ Redundant Code

As above-mentioned, removing dead code can significantly improve a program quality. The amount of dead code can give a quantitative data shown the value of a program. If the amount of dead code is significantly more than the operative code, it has better to be re-designed, so that the maintainability of the source code can be improved.

Difficulties of implementation

Same as above-mentioned, to detect dead code in PL/I, it has to be compiled. It is hard to detect dead code, let alone the lines of dead code. However, the local dead procedure can be detected. The LOC can be known in the dead procedure. Nevertheless, same problem happens here, it is only for the local procedures that are not called by the local programs. It is still an open issue in this project.

***Cloning Code Detection**

Cloning code is some duplicated code fragments that is still widely used in reuse strategy. It happens relatively more in PL/I programming, because the abstraction mechanisms is not that sophisticated as Java or any other object-oriented languages. It often leads to a plenty of duplicated code in a large software systems like the one in BMW. However, cloning can cause a lot or problems in software maintenance for several reasons.

Firstly, cloning increases program size. The larger program size, the more maintenance efforts is needed. To duplicate a piece of code, maybe it is the fastest way to duplicate the code logic. However, it may costs more maintenance efforts to maintain it, like read and understand the code. Secondly, the changes of one clone probably needs the other changes in the other clones. For example, for bugs fixing, not only one place to be fixed, but also another places that is cloned are needed to be fixed. Finally, the changes of duplicated code can possibly introduce bugs in a software project. Normally, people do not have to 100% clone, but with some minus changes which is specific for the purpose on their programming design. However, changing duplicated source code fragment may lead the performance inconsistently. Therefore, clone is not a good practice in programming.

Implementation

For this rule, due to the complexity of implementing the clone detection algorithm, it is better not to implement it from scratch. Therefore, in the application, ConQAT clone detection [26] will be integrated into the PL/I checkstyle application. The method that ConQAT are using is an efficient detection algorithm, which is using Abstract Suffix Tree (AST). The computational time complexity is just O(n). Therefore, it is better to use their implemented clone detection algorithm which is the fastest way to detect cloning code. Although the clone detection algorithm does not need to be implemented, the integration is not trivial. First of all, a configuration XML file that contains the information of input directory of PL/I files, the output directory and clone detection setting is needed to be generated. This is triggered off when the users start the clone detection. And then the configuration file will go the ConQAT wrapped by a ServletDriver. This ServletDriver makes ConQAT easily to be triggered by another program. ServletDriver gets all block files and configuration files and then distribute them to the corresponding classes. After finish the clone detection, it will generate a clone report which is in XML format.

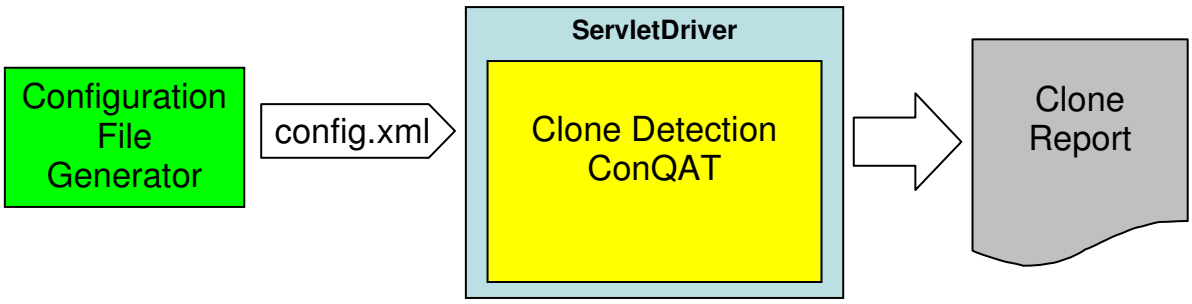


Figure 4.2 Flow chart of clone detection

***Nested Loop Depth**

A nested loop is a loop within a loop, an inner loop within the body of an outer one. This is a common practice in the most programming situation. However, an over-nested loop would cost much effort to read and understand. Also, a small change inside the nested loop group may introduce bugs which are probably not obvious to figure out. Therefore, it is not a good practice to have an over-nested loop. However, it is hard to justify how many nesting level should not be allowed. It is because it is depending on the project needs. For example, for some numerical simulations, they might need some over 3 nesting level loop group to calculate the data. It is difficult to change their algorithm to become less than 3 nesting level loop but without changing the output. Although the nested loop depth is important information to tell the quality of a program, it is still hard to define which nesting level is over-nesting. Nevertheless, nested loop depth is critical to know if the quality of program is good or not.

Implementation

In PL/I, DO statement and its corresponding END statement, delimit a group of statements collectively called a do-group. If there is an expression within a DO statement, it will become a looping group to provide for the repetitive execution of the statements within the do-group. Normally, by identifying the expression, we can see that if the do-group is a loop or not. First step, DO statement has to be located. And then, find out its corresponding END token and check if the DO statement has an expression. If there is an expression, this is a looping do-group. Second step, find if there is another looping DO-group inside the previous group. If so, do the second step again. Otherwise, find another DO-group in the rest of the source code.

Nested Loop Depth across Procedure

A nested loop across procedure means an executive statement calls a procedure which has nested looping group. And this executive statement may or may not be wrapped by a nested loop group. For example, an executive statement wrapped by two loops calls another procedure which has a loop group with 3 nested levels. This means the total number of loop depth is 5. This information is also critical, because when programmers design their programs, they would not consider the loop depth across procedure normally. However, it does take significant computational time as much as nested loop without across procedure.

Difficulties of Implementation

It is difficult to know the relation between the procedures until the program is compiled. And also, in the BMW mainframe system, programmers do not have the whole set of software system in their local computers. Therefore, in this case, this is hard to be completely implemented. Nevertheless, the local call graph can provide some considerable information of the local program call relation. A procedure which calls another procedure normally has to declaration of the external procedure that is going to be called. By identifying the external procedure declaration, it is possible to know the relation between procedures. Moreover, checking loop depth level of CALL statement, it is one of the ways that counts the total nested loop depth across the procedures which are in the same source files folder.

***OTHERWISE case in SELECT statement**

SELECT statement in PL/I is same as the SWITCH statement which is the counterpart of Java. The OTHERWISE case is same as DEFAULT case in SWITCH statement in Java. Normally, a SELECT block is still able to be compiled even without OTHERWISE case. However, programmers might overlook the exceptional case happened. It can introduce potential bugs in a program. Hence, by adding OTHERWISE case in each SELECT block is a must during programming, so that reduces the possibility of compilation error or runtime error. Therefore, it is one of the rules that is important for PL/I programmers.

Implementation

To implement this rule, a small SELECT group parser is needed. First of all, by scanning through a source code, SELECT group can be identified and located. It is all done by a SELECT group parser. And then in each SELECT group, we have to check in this group region if there is a keyword OTHERWISE. If so, this is a qualified SELECT block; otherwise, the programmer will get a warning of SELECT block without OTHERWISE case. For implementing the SELECT group parser, there is a non-trivial point that has to be paid attention at. It is because SELECT is also a keyword of SQL statement, so we have to distinguish whether it is a PL/I or SQL keyword. For the case of SQL statement executed in PL/I program, normally there

would be two keywords have to be stated before the SQL statement. These two keywords are EXEC and SQL. Therefore, we have to check if there are these two keywords before SELECT statement, so that we can distinguish them.

***LOC within WHEN case in SELECT statement**

WHEN in SELECT block is same as the CASE in SWITCH. In each of the WHEN case, it should not be too many LOC. In each WHEN case, it is better to use as less LOC as possible. It helps programmers to read and understand the code. Also, a lot of codes in each of the WHEN case would not be a good structure of a program. It is better to encapsulate the code inside the WHEN case if there are many LOC.

Implementation

In PL/I, it allows only one statement in each WHEN case. If programmers need to put more statements to be executed inside in each WHEN case, block is needed. Block can be a DO-group, BEGIN-group, or even another SELECT group. Therefore, by detecting a BLOCK inside a SELECT group, it can tell the LOC within WHEN case. A small block group parser can find out all groups in a program. Hence, by looking for the block in a SELECT block, it can tell the LOC in WHEN case. Of course, all the blocks in a SELECT block have to be checked if they follow the keyword WHEN or OTHERWISE, so that it makes sure that they are the blocks in case.

***Code margin**

For the input text of the PL/I program, the document margin is restricted between columns 2 and 72. If the code is out of the margin, the code will not be compiled. However, programmers may not receive error message or warning message from the PL/I compile. Therefore, the rule helps programmers to locate the potential bugs in a program.

Implementation

By using Lpex Parser, it can check the margin problem. Going through every position in a program text document, checks if there is a text or token in that particular position. When the position is out of the margin and there is a text or token, this means the code margin restriction is violated.

***LOC in ON unit**

ON-unit specifies the action to be executed when the condition is raised and is enabled. The action is defined by the statement or statements in the ON-unit itself. When the ON statement is executed, the ON-unit is said to be established for the specified condition. The ON-unit is not executed at the time the ON statement is executed; it is executed only when the specified enabled condition is raised. The ON-unit can be either a single unlabeled simple statement or an unlabeled begin-block. If the ON-unit is a BEGIN-block or DO-block, it can contain a set of statements are going to be executed. However, due to the complexity of this condition handling, it is better to make it as simply as possible. Therefore, LOC in ON-unit can tell the complexity of the operation of ON-unit.

Implementation

It is done by block-group parser which locates all the blocks and groups in a program. It extracts all the tokens in the program. If ON statements is encountered, it will be checked if there is a block or a group starting within the ON statement. And then it is compared to the block or group that if it is matched with the block or group location computed by block group parser. A group and block can be easily computed their LOC by comparing block/group start and end tokens.

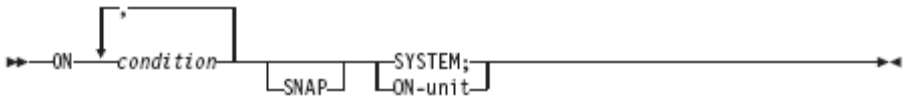


Figure 4.3 Abbreviation ON statement

***Number of ON-unit and their locations**

As above-mentioned, ON – unit establishes the action that is to be executed whenever the enabling condition is subsequently raised while within the cope of the ON condition. However, to know when the condition has been enabled, it would be very difficult task. It is because the condition in a procedure is raised in implicit way. Therefore, once many ON statements have been stated, these ON statements would increase the complexity of the code.

Implementation

It can be done by simply checking the ON token in a program. By extracting the ON-token and distinguishing them between ON-statement (i.e. only one statement (action) is executed) and ON-unit (i.e. followed by a block or group which embraces a set of statements or actions), the total number of ON-unit and On-statement can be found and their locations can be known.

***Number of parameters passing into procedure**

A procedure in PL/I is similar to method in Java. Normally, they can either return a value or just return null. By passing parameters into a procedure, different results will be calculated. In this rule, the number of parameters passing into procedure would be interesting information that justifies whether a procedure is well-defined. It is because a procedure with a long list of parameters would not be in a good abstraction. Also, it would be easily to make mistakes while calling this kind of procedures, like missing parameters or wrong positions of parameters. Therefore, the number of parameters to be passed into a procedure should not be too many e.g. more than 5 parameters. It is better to wrap the procedure by another procedure, so that reduces the number of parameters list.

Implementation

To get the number of parameters of a procedure, it can be done by analyzing the procedure statement. As shown in figure 4.4, in a procedure statement, the parameters list is located right after the keyword PROCEDURE. Therefore, once PROCEDURE token is located, the parameter list with a bracket can be analyzed. By looping through the bracket and counting the number of comma inside the bracket, the number of parameters can be found.

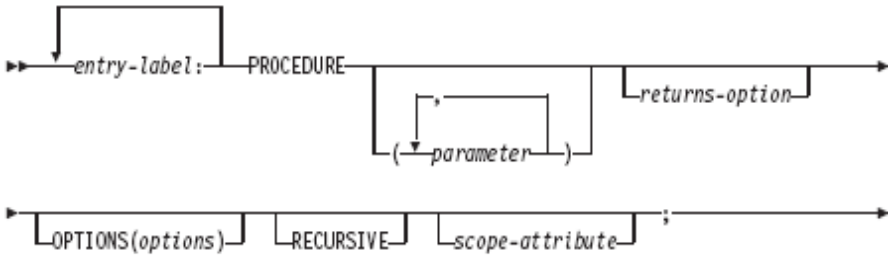


Figure 4.4 Abbreviations PROCEDURE

Stopping criteria in recursive procedure

One basic form of recursive procedure is to define one or a few base cases that recursively call the procedure itself, and then define rules to break down other cases into the base case. However, if the recursive procedure is failed to define the condition that stop the recursion, it will happen error during the runtime, like dead-loop etc. By identifying the stopping criteria in a recursive procedure, it will help to minimize the opportunity to get error in a program.

Difficulties of Implementation

To implement this rule, a recursive descent parser is needed. However, in PL/I, to implement this parser is quite difficult, because of PL/I special feature, for instance, keywords are not reserved which makes the parser much more complicated to be implemented. Also, the time is another restriction due to many rules having to be implemented. Therefore, this rule will be taken as a future work.

***Foul Language Checking**

Insulting words in a source code is not good manner and have no advantage in development. However, sometimes, it still happens in code comments or printing text. Foul language in comments or in printing text means that they are meaningless. They can explain nothing about the codes. Therefore, in order to eliminate redundant and meaningless content in a program file, foul language should not be allowed.

Implementation

In the case, German and English foul language is checked. All these foul words are specified in a text file which can flexibly add or remove the words. When the foul language analyzer is triggered, all the foul words in the text file are read and stored in a Hashmap. And then, by using Lpex parser, checks the each of the words in the program file and compares them to the foul words Hashmap. Foul words in comments can be found.

For the case of foul words in a string, PL/I token scanner has to be used. By scanning through the program file, locates the strings and splits the string up to put these substring into an array of strings. By comparing the array of strings to the Hashmap of foul words, the foul words can be located.

***Block/Group in Preprocessor**

PL/I preprocessor is a very complicated and powerful feature. Using preprocessor can simplify parameterization of a program. Also it will allow the compiler to generate the most efficient code. It even can modify the code during the compilation time, i.e. by macro facility. Therefore, in order to minimize the code complexity in using preprocessor, it had better not put a block or a group in a preprocessor, but encapsulates the block of statements in a procedure or a file. This is much easier to read.

Implementation

A block or a group can be found by block-group parser. And then, what we have to do is locate the preprocessor statement. If there is a block or a group started in a preprocessor statement which is matched with the output of block-group parser, it can be regarded as a block or group in a preprocessor. However, due to complexity of preprocessor feature, for the case of using macro facility, it is still difficult to be checked.

***Number of Block/Group**

Although the number of Block/Group cannot tell the size of the whole program, it can tell some information about the structure of the program and locate the blocks and groups. For example, if a program file has about 1000 LOC, but there are only 3 blocks and groups in total. Would it be good structure of the program? The answer must be negative. On the other hand, a file with only 100 LOC has 50 blocks and groups in total. There would be a question - “Is there any empty blocks? Or some blocks are doing nothing”. This also can tell structure information of a program.

Implementation

Actually, in order to get the number of blocks and groups, it is based on a Block-Group parser. This Block-Group parser extracts PACKAGE, BEGIN and PROCEDURE as block and DO and SELECT as a group. By locating and counting the block or group started keyword above and END, the block and group can be found out; meanwhile the number of blocks and groups can be found. However, this implementation cannot automatically optimize when multi-end block happen (i.e. a block with more than 1 END statement). If multi-end block or group happens, this would be regarded as a failed block or a failed group.

***Block-Group Depth**

Block-Group Depth refers to their block-group nested depth level. Block-Group depth can also provide structure information of a program. As shown in figure 4.5, a nested block structure is defined in this way. The outermost block is level 1 –L1 and it can have infinite inner block level n –Ln. The nested block is quite hard to read and understand. Therefore, the over-nested block-group depth level should not be allowed.

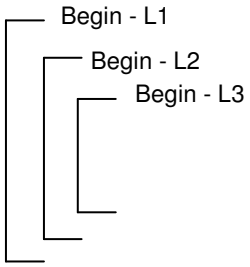


Figure 4.5 Block-Group Depth Definition

Implementation

Firstly, get all the blocks and groups from a program file by using block-group parser. Normally, the output of the parser contains the block / group starting and ending tokens. By comparing their positions of start and end tokens, it can simply identify their depth level.

***LOC in Block-Group**

LOC in a block or group can tell the size of the block or group. A proper size of block and group can help programmers to maintain the code.

Implementation

It is very simple to implement by using block-group parser. The output of the parser would be a pair of tokens which are start and end of the block or group. The LOC can simply be calculated by getting the difference between the start and end token location.

***Failed Block-Group**

Failed block and group means a block and group do not have an appropriate number of END token. For example without END token or more than one END token (i.e. multi-end block). It would be regarded as a failed block or a failed group. In the case of multi-end block, the compiler may not give an error message. However, for the software maintenance point of view, it is confusing practice.

Implementation

The implementation can be done by checking a block or a group if they are ended with one END token. If not, the block or the group will be regarded as a failed block.

***Empty Block-Group**

Empty block or empty group can be regarded as a redundant code, because, in an empty block or group, there is nothing to be executed. It is not a good practice to put a piece of code in a program but without executing anything. This confuses the programmers who maintain the code. On the other hand, it makes the code size unreasonably increase.

Implementation

The block-group parser is used to get the block and group information from a program. As above-mentioned, the output of the parser is a list of block and group information which contains blocks or groups starting and ending token. By checking if there is no statement between the starting token and ending token, empty block or empty group can be found. It also can be done by checking if there is LOC between starting token and ending token and if there is a token between the block start and end.

Block/Group Indentation Consistence

Indentation is one of the programming styles in most of the programming language. In order to increase the readability of the program code, block or group indentation consistence can help a lot. Indentation is depending on the block depth level. In the same depth level, the indentation should be consistent.

Difficulties of Implementation

To implement indentation checking, it needs to identify how much indentation corresponding to block and group depth level is needed. Firstly, the program has to be scanned through. And then, the block and group structure of the program is identified. However, by comparing the indentation of the blocks that have the same depth level, it costs a lot computational time. Also, it might have warning message flooding problem, especially the indentation warning. Some quality checking applications also have the same problem. Almost every single line has the indentation problem. Meanwhile, programmers cannot fix all these indentation very easily. It would become an annoying message but a productive message. Therefore, the indentation warning checking rules still have rooms to be improved. For example, if the Lpex editor provides functionality that automatically fixes the indentation problem, like the Java editor in Eclipse, it does really help the programmers, because the indentation problem can be fixed very easily by the indentation automatically fix.

***Calling Relation of Procedures**

Calling Relation between procedures shows how the procedures call each others. It can help programmers to understand the programs and the results can help to have further analysis of the programs. To show the analysis results of calling relation of procedures, a tree view is used to present the results. It is similar to call graph that shows a hierarchy tree that show callers and callees. Although it is not like the call hierarchy for Java in Eclipse, it still can provide information to analyze the programs, for instance, it can help to analyze the nested loop across procedures and dead procedure (i.e. procedure that are not called by the others) etc. The tree mainly shows the information of file, procedure and called procedures. The parent nodes are the PL/I files that are analyzed. And each of the file nodes would have their own procedures which is the child node. Within the procedures, it contains the nodes of the called procedures. Based on this tree, it can have more information to analyze the codes.

Implementation

As shown in figure 4.6, First of all, a single file or files in a folder is/are scanned. By scanning through the files, internal procedures, external procedures and the procedures in the files are found. The internal procedures mean the procedures inside other procedures, so the procedures can call its internal procedures freely. External procedures mean the procedures exist separated with other procedures. If procedures need to call these external procedures, the external procedures have to be declared before it being called.

Secondly, based on the information by scanning through the files, a XML file will be generated which records all the tree view information. For the analyzers that need to use the information in this XML file, there is a class which can let programmers read the specific data that they are interested.

Finally, by reading this XML file, the calling relation tree view can be created.

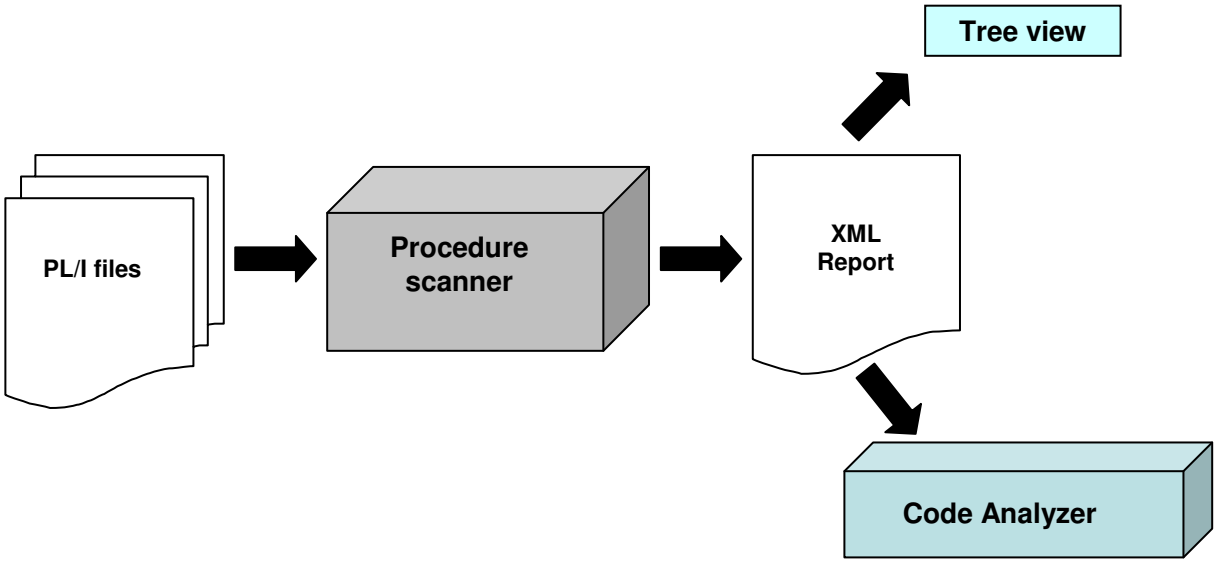


Figure 4.6 Workflow of calling relation of procedures

***Number of being called of a procedure**

A number of being called of procedure can show the reusability of the procedure. Also, it can help programmers to re-organize the structure of a program, based on the number of being called. On the other hand, this information can show the dead procedure (i.e. a procedure not being called by other procedures). For this kind of procedures, it had better remove it, so that makes the code become more maintainable.

Implementation

By simply getting data from the XML file of calling relation tree, the number of being called of a procedure can be counted by going through all the elements in the XML file.

***Number of external procedures**

The number of external procedures in a file can show how many procedures have to be imported into a file. The number can tell how many procedures are depended by the files.

Implementation

By simply scanning the file, if the procedure is external, the declaration is needed. It would like this statement – "`DCL procedure1 EXTERNAL ENTRY;`" Therefore, by recognizing the keywords `EXTERNAL` and `ENTRY`, it is good enough to identify the external procedures. On the other hand, it also can be done by scanning the XML file from calling relation tree view. This also provides information of external procedures of each file.

***Loop Depth Level of CALL**

This checks the loop depth level of CALL statement or a function triggering statement. It gives information of loop depth level of CALL statement or a function triggered statement. For example, a call statement with 3 loops wrapped and every loop with O(n) computational time complexity, the total number of executing the statement would be $O(n^3)$. If the procedure or function called with another nested looping group, it would make the computational costly. Therefore, this can give some clues on the programs performance.

Implementation

It also can be done by reading the XML file from calling relation tree. In the XML file, it locates the loop depth level of all procedures that are called within a procedure.

***IF-ELSE Block**

Actually, there is no IF or ELSE block in PL/I. There is only IF statement. As shown in figure 7, it shows the structure of IF statement. Nevertheless, by identifying IF-ELSE block, would help to analyze the code. It can provide information of nested IF statement which is also not a good practice from software maintenance point of view, because it also consumes time to read and understand.

In this case, it would be two kinds of block. IF block and ELSE block. As shown in figure 4.7, there are unit 1 and unit 2 which are used to distinguish between IF block and ELSE block. IF block includes IF, expression, THEN and unit 1. ELSE block includes ELSE and unit2. Normally, unit1 and 2 is a set of actions that executes when the corresponding condition expression is raised. And each of the unit can have another IF block. However, for ELSE block, it cannot have another ELSE block without IF block. Otherwise, it will have compilation error.

Figure 4.7 Abbreviation IF statement

Implementation

In this case, IF-ELSE block parser is needed to be built. To build this parser, the PL/I token scanner scans all the tokens in a file. And then, the block is identified in the unit1 and unit2. This parser gets IF-unit, ELSE-unit and ELSE-IF unit. IF-unit means the block started from IF token to unit1. ELSE-unit means the block started from ELSE unit2. ELSE-IF unit means the block started from ELSE and IF tokens to unit2. The information can be used by other analyzers.

***Depth Level of IF statement (nested-IF statement)**

Nested-IF statements group has the same problem with nested loop group. Although it does not consume computation time, it does consume maintenance time from the software maintenance point of view. It is not very easily to understand such a nested-IF statements group, for instance a nested-IF group with 4 nesting level. However, similar to nested loop case, it is difficult to justify whether the number of nested level is good or bad. It is dependant upon the need of software project. However, it is a critical information that shows the complexity of a group of IF statement.

Implementation

By using the result of IF-ELSE parser and comparing the location of the start and the end of the block, the depth level of IF statement can be found.

***IF statement Ratio**

IF statement ratio is calculated by the number of IF statement divided by the total number of statement in a source file (SLOC). This information can give an idea that the complexity of the usage of IF statement in the program file. For the case of IF statement is too many comparing to the SLOC, it reflects that the program file is complicated to be maintained. It is because people cannot easily understand mechanism of the IF statement, especially some complicated expression. From the software maintenance point of view, it also consumes time and manpower to maintain a source code with major portion of IF statement.

Implementation

As above-mentioned, by counting number of IF statement and total number of statement in a source file, the ratio can be achieved. To count the IF statement, a file has to be scanned through by PL/I token scanner. And then, by simply counting the IF token, it can be found out the number of IF statement.

***Number of ELSE IF in a group**

Normally, ELSE IF structure has the same operation with SELECT group. Therefore, the number of ELSE IF in a group can tell the complexity of the IF-ELSE IF statement operation. If the number of ELSE IF in a group is too many, it had better use SELECT group. It is because it has less code to type. Moreover, the structure of SELECT group is much clearer. Therefore, this number is critical to tell programmer whether they should choose a better structure to program.

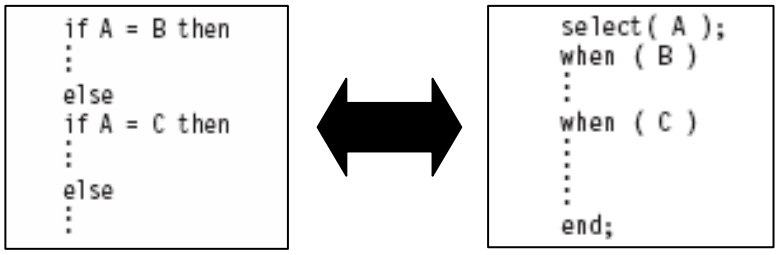


Figure 4.8 Comparison of ELSE-IF and SELECT group

Implementation

To implement this checking, the IF-ELSE block parser is needed. Since the parser can extract the ELSE-IF unit, the number of ELSE-IF can be done by counting the number of ELSE-IF unit. However, in order to know whether ELSE-IF units are in the same group, their location has to be checked. And check if there is another IF unit between the ELSE-IF unit.

***Bit Operation in IF expression**

The complexity of IF-expression can also consume more effort in software maintenance. The more bit operations in IF-expression makes the expression more difficult to read and understand. A bit operation is specified by combining operands with one of the following logical operators: `&, |, ^.` Therefore, by counting the number of the bit operation in the IF-expression, the complexity of IF-expression can be known.

Implementation

First of all, IF statement has to be located by using PL/I token scanner. And then the number of bit operation can be counted by checking through the IF statement until the token of THEN.

***Comparison Operation in IF expression**

As above-mentioned, the complexity of IF-expression can be quantified by the number of logical operation within the IF-expression. Similar to bit operation, comparison operation is also one kind of logical operations in the IF-expression. A comparison operation is specified by combining operands with one of the following infix operators: `<, ^<, <=, =, ^=, >=, >, ^>`.Therefore, counting the number of comparison operation in the IF-expression can also show the complexity of IF-expression.

Implementation

Similar to bit operation, the number of comparison operation can be counted by checking the tokens between token IF and token THEN.

***Blank line ratio**

Blank line ratio is calculated by number of empty lines divided by total number of LOC. An appropriate amount of blank line within a program code can enhance the readability of the code. It is also one of the programming standards in Java. The readability of PL/I program can also be improved by obeying this standards. Although this information cannot check the position of blank line, it can give an overall idea the readability of the program code.

Implementation

It can be done by scanning through all lines in the files to check if there is a non-commented token in the line. If a line without any tokens, it would count as a blank line. And then, the blank line number is divided by the total LOC, so that the blank line ratio can be achieved.

***One statement per line checking**

This rule also helps to check the readability of a program file. This rule restricts programmers to put only one statement in one line. It would not be a good idea to put many statements in one line, no matter how long the statements are. This rule can make sure that the program layout structure is much clearer.

Implementation

Since a statement in PLI must be ended with semicolon, by simply locating the semicolon, this rule can be checked.

***SELECT expression data type checking**

The data type of expression checking prevents from the error during execution time. It is because the PL/I compiler does not give an error message of inconsistent data type of the expression in SELECT group. Therefore, this checking can help to avoid error occurring during the runtime.

Implementation

By locating the declaration statement of expression data type, the data type of expression can be checked. The expression in SELECT statement must be declared before SELECT group is called. However, for the expression in WHEN statement may not need to be declared. For example, if the data type of SELECT expression is character, in the WHEN expression, it is possible to put a character, like ‘a’ and ‘b’. In this case, the PL/I token scanner is needed to get the type of token which is in the WHEN expression, so that it can compare to the type that is in SELECT expression.

Chapter 5

Architecture of PL/I Checkstyle Application

This chapter presents the architecture of PL/I Checkstyle application. The PL/I Checkstyle application can be divided into two parts – Checkstyle Platform and PL/I Analyzers. Their core components and the processing steps will be presented. Also, the Checkstyle platform has extensible feature which allows adding new analyzers flexibly by using Plug-in mechanism.

5.1 Overview

The architecture of the checkstyle application is shown in figure 5.1. It is mainly divided into two layers: Legacy Checkstyle Platform and the set of Analyzers. The Checkstyle application is an Eclipse Plug-in application that working principle is based on the Plug-in mechanism.

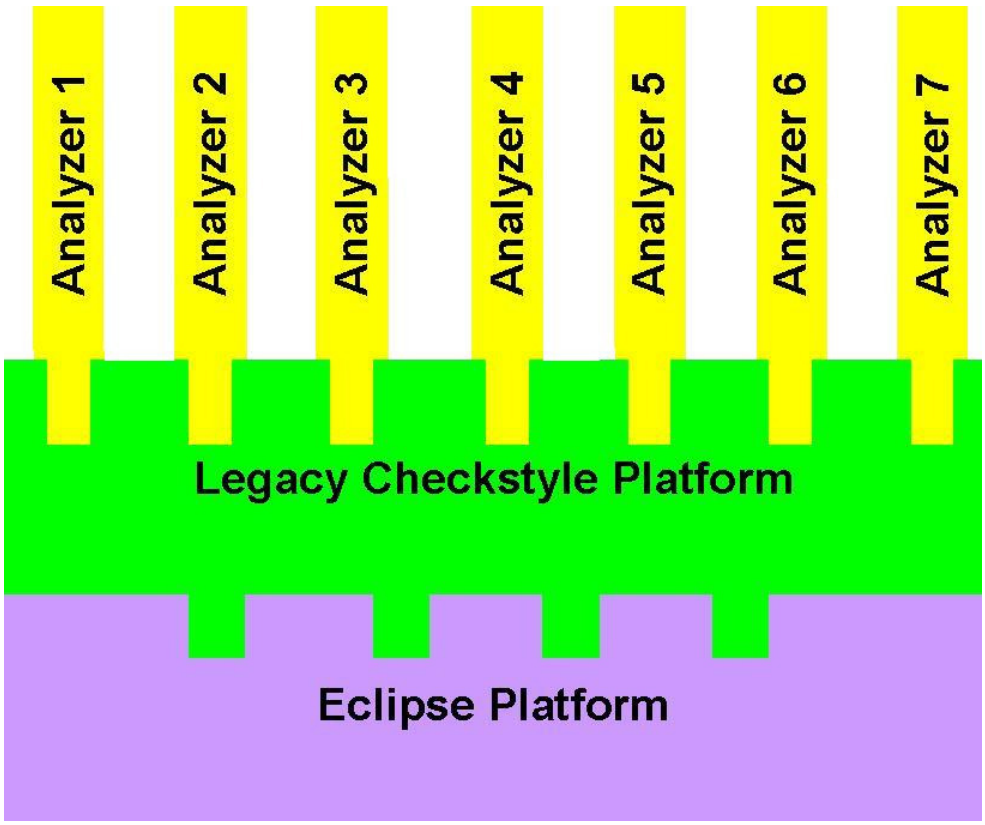


Figure 5.1 PL/I Checkstyle Architecture

The following figure is the package and class diagram of the Checkstyle platform and the Analyzers. This diagram shows the dependencies of these two parts.

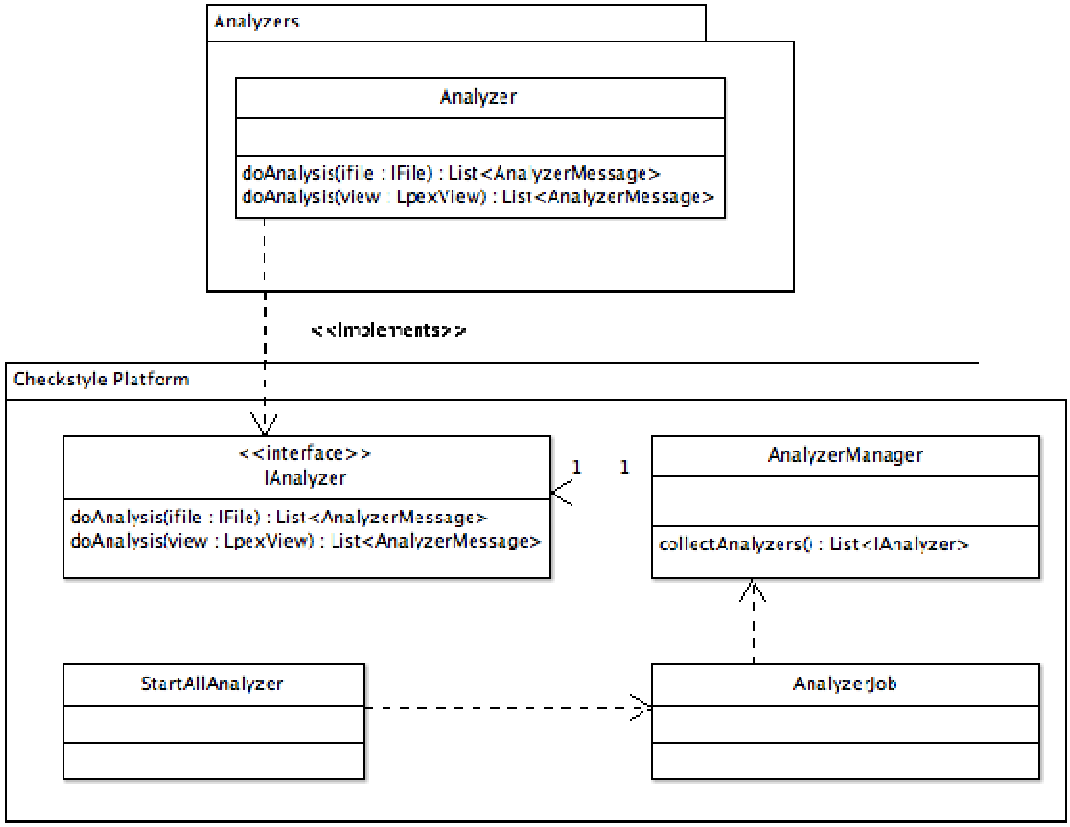


Figure 5.2 Package and Class diagram of PL/I Checkstyle application

Legacy Checkstyle Platform:

- The platform plugs into the Eclipse Plug-in.
- The platform provides extension point for the Analyzer Plug-ins. The analyzer can easily plug in to the Checkstyle platform and run it.
- Multi-language code analyzer is supported. This means the Checkstyle platform can support different programming language code analyzers.
- The platform is responsible for results presentation and getting source files.
- Useful Tools are provided by the platform. For example, PL/I token scanner, Calling Relation of Procedure analyzer and Clone Detection are in the platform. They can provide information for the analyzers.

Analyzers:

- PL/I code analyzers
- They Implements a Java interface – IAnalyzer.
- It can be any kinds of programming language analyzer.

5.2 Working Process of PL/I Checkstyle Application:

The working process of PL/I Checkstyle application can be divided into 3 parts. Since the application has mainly 3 analysis processes. They are PL/I Checkstyle, Procedures analysis and Clone detection. Since they are using different data model structure and different architecture to implement, the working processes of them are quite different. Also, the working process of preference page is also shown in this section.

5.2.1 PL/I Checkstyle

The working process of the PL/I Checkstyle is a process that includes getting the source files, running code analyzers and displays the results. The figure 5.3 shows the activity diagram of PL/I Checkstyle.

In the beginning, the Plug-in packages are launched. Programmer starts PL/I Checkstyle with selecting the target folder with PL/I files or a single PL/I file. The Checkstyle platform then starts to load the analyzers, which is regarded as a Plug-in package.

Based on the Plug-in mechanism in Eclipse, the analyzers that are implemented are loaded by the Checkstyle platform. The analyzers are just listed in Plug-in XML file, which is defined by the programmers who extend the code analyzers. When PL/I Checkstyle is started, the analyzer classes are loaded and stored into a list for the next activity.

After all analyzer classes are stored into a list, it is the time to run them one by one. Each of the analyzers has an interface to be implemented. By calling this implemented interface, the analysis can run.

The output of each analyzer would be a list of analysis result messages - `List<AnalyzerMessage>`. `AnalyzerMessage` is an object that contains the information of violation of the rules, the value of the output and the location of the problem.

The user can stop the process whenever they want. If the user stops the process while the analyzers doing the analysis, the results of the analyzers that have already finished are still stored in the list of `AnalyzerMessage`. Therefore, the part of results can still be achieved.

After getting the results from the analyzers, the results are transformed into different format to display. There are a problem view and info view to display the results. Also, the results would be transformed into various data formats that can make the

presentation of the results become much more interactive with the users. For example, Marker can make the results interacted with the text editor.

In each of the transition, it is possible to get the error, for example, during the activities of getting source files and loading the analyzers by Plug-in mechanism. Therefore, the error happening during the process will be caught. And the error is handled by throws exception mechanism.

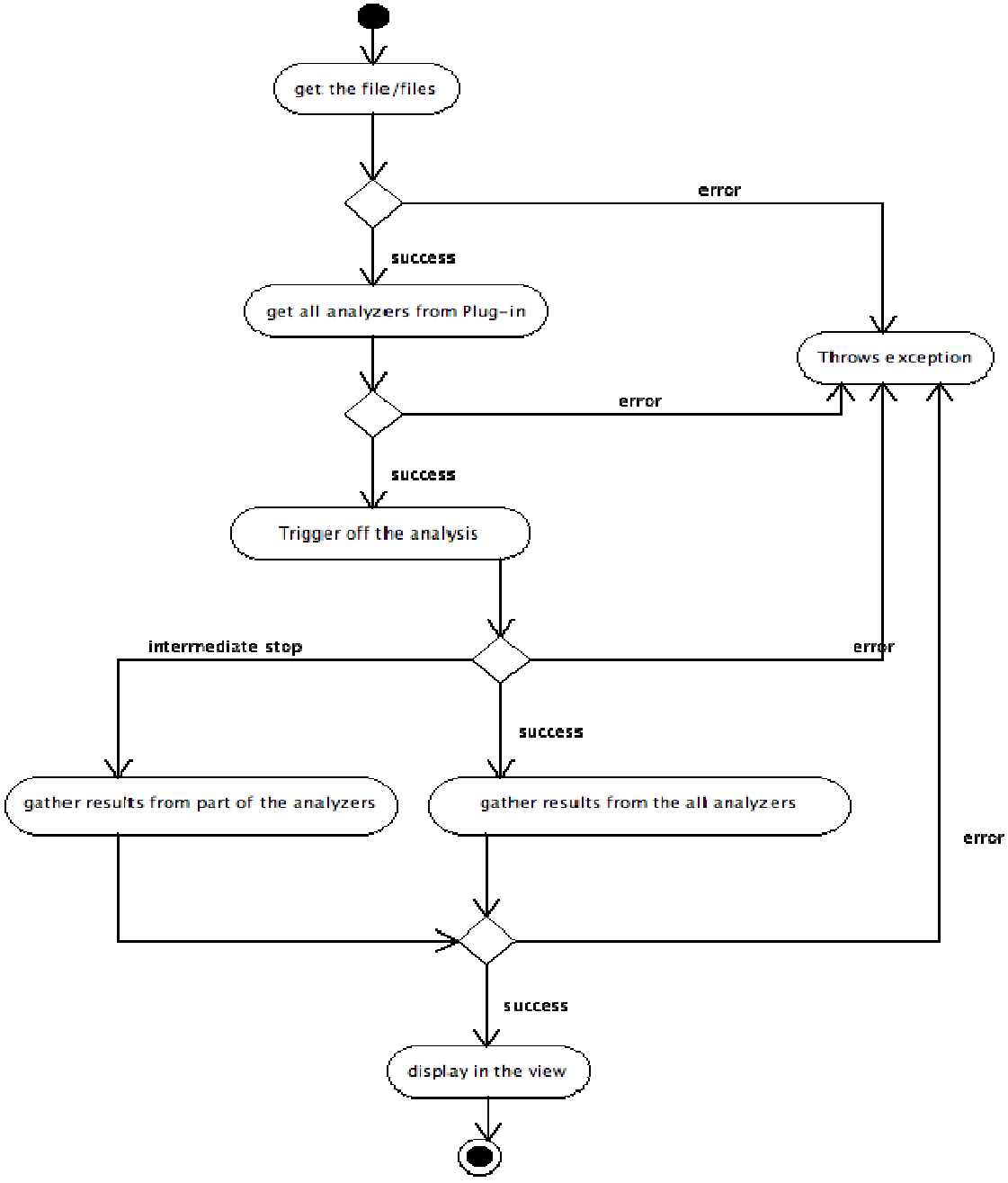


Figure 5.3 Activity diagram of PL/I Checkstyle

5.2.2 Procedure Analysis

Procedure analysis can provide information of calling relation between procedures or function. The working process of procedure analysis includes getting the files, getting the procedures and analyzing the calling relation of procedures. The figure 5.4 shows activity diagram of procedure analysis.

Firstly, same as PL/I Checkstyle, user starts procedure analysis with selecting the target folder with PL/I files or a single PL/I file. If the user selects a single file to analyze, it would only analyze the procedures in this file. If the user selects a folder that contains the PL/I files to analyze, it can analyze the procedures in these files.

After getting files, each of the PL/I files will be scanned and the procedures and functions are extracted. Procedure includes internal procedure and external procedure. An internal procedure refers to the procedures that are inside of the file. An external procedure refers to the procedures that are outside of the file. By getting these two kinds of procedures, it prepares for the next activity to analyze.

To do procedure analysis, each of the procedure blocks has to be scanned through. Searching and locating (i.e. Loop depth level of calling statement) the statement that calls another procedures can find the relationship between the procedures. Also, getting the data about procedures can provide information for another analyzer to use.

And then, based on the result of analysis, it will generate a procedure report to record the result. The report is in XML file format. This XML report file provides data model and information for the result presentation and the analyzers that are created in PL/I Checkstyle.

Finally, after the XML file report generated, A XML reader will read the file and generate a tree view to display the calling relation between procedures.

It is also possible to get error in each of the activity transition. Therefore, there is an error handler to catch the all the errors. The error is handled by throws mechanism.

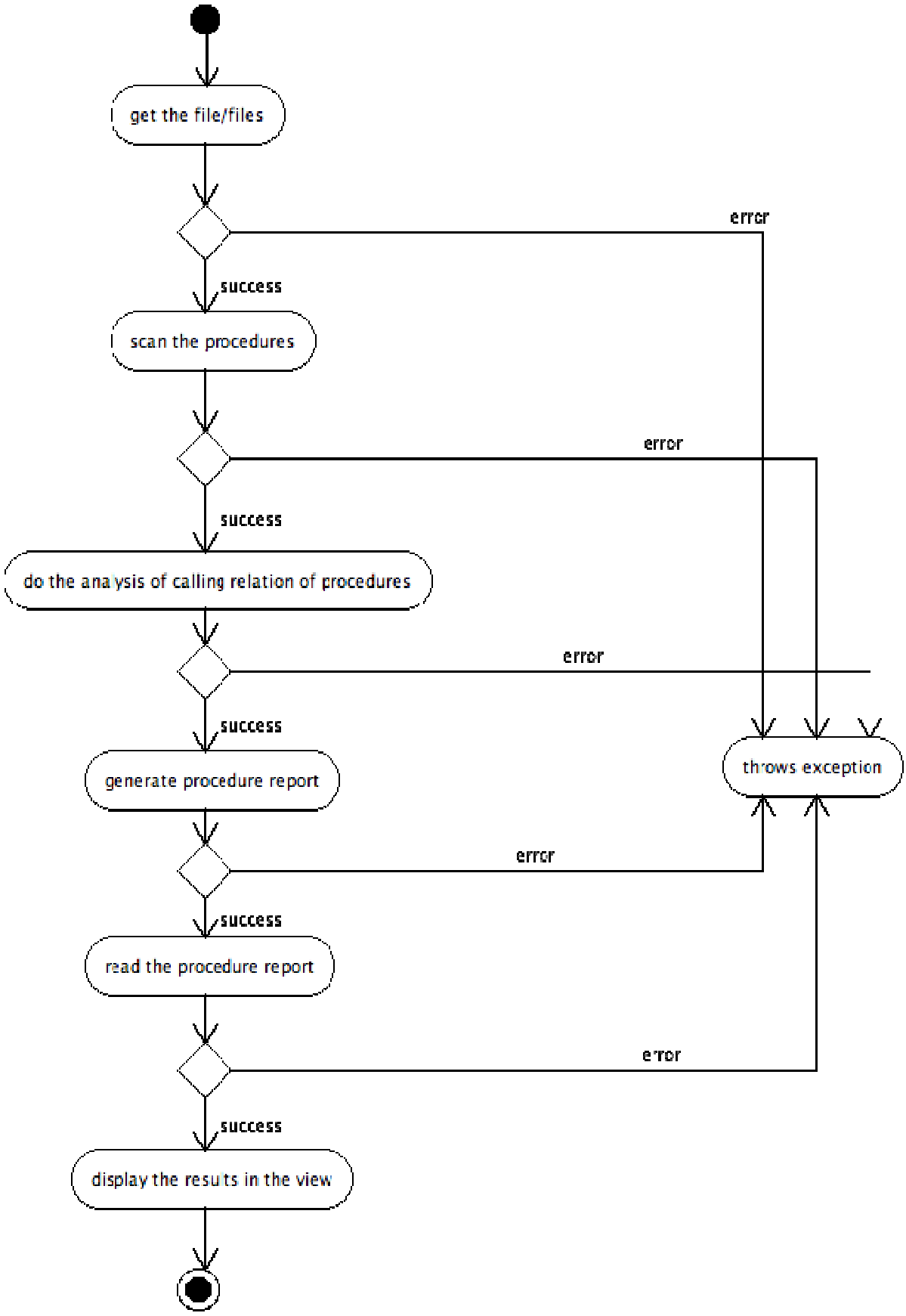


Figure 5.4 Activity diagram of Procedure Analysis

5.2.3 Clone Detection

Clone detection checks the duplicated codes. The working process can be divided into 4 parts- generating configuration file, starting clone detection, generating clone report and reading the clone report to present the results in tree. Since the core clone detection package is from ConQAT [27], the detailed algorithms of clone detection will not be discussed in here. The figure 5.5 shows the activity diagram of clone detection.

Firstly, user starts the clone detection by selecting the target folder. Based on the selection by the user, a configuration file will be generated. This configuration file configures the clone detection settings, for example, the normalized length, input and output folder etc. Nevertheless, the settings are fixed (apart from the input folder), so that users would not need to bother about the configuration of Clone Detection. It makes the functionality much easier to use.

Secondly, after the configuration file is created, the clone detection can be started. The clone detection would run based on the configuration file settings. The clone detection uses a sophisticated algorithm by using abstract suffix tree, which can detect fuzzy clone and strict clone.

After the clone detection done, it will generate a clone report, which is in XML format. The output information in the clone report can also be modified by the configuration file. In this case, we only need the location of the clone, size of the clone and number of the clone. In the report, it contains clone-class, which specifies the code cloning information in the PL/I files. Clone-class contains a certain number of clones. In each of the clone, it gives the information of length of code cloning and location of the clone. Also, this clone report can provide information for extending analyzers.

By reading this clone report, the information of clone report is presented in a tree view. Since XML file has a tree-liked structure, the tree view can take most of the information and data structure from the clone report.

Finally, error is possible to happen during the process. It also has an error handler to handle all errors. It can tell the user where the error is, so that it helps user to cope with the problem.

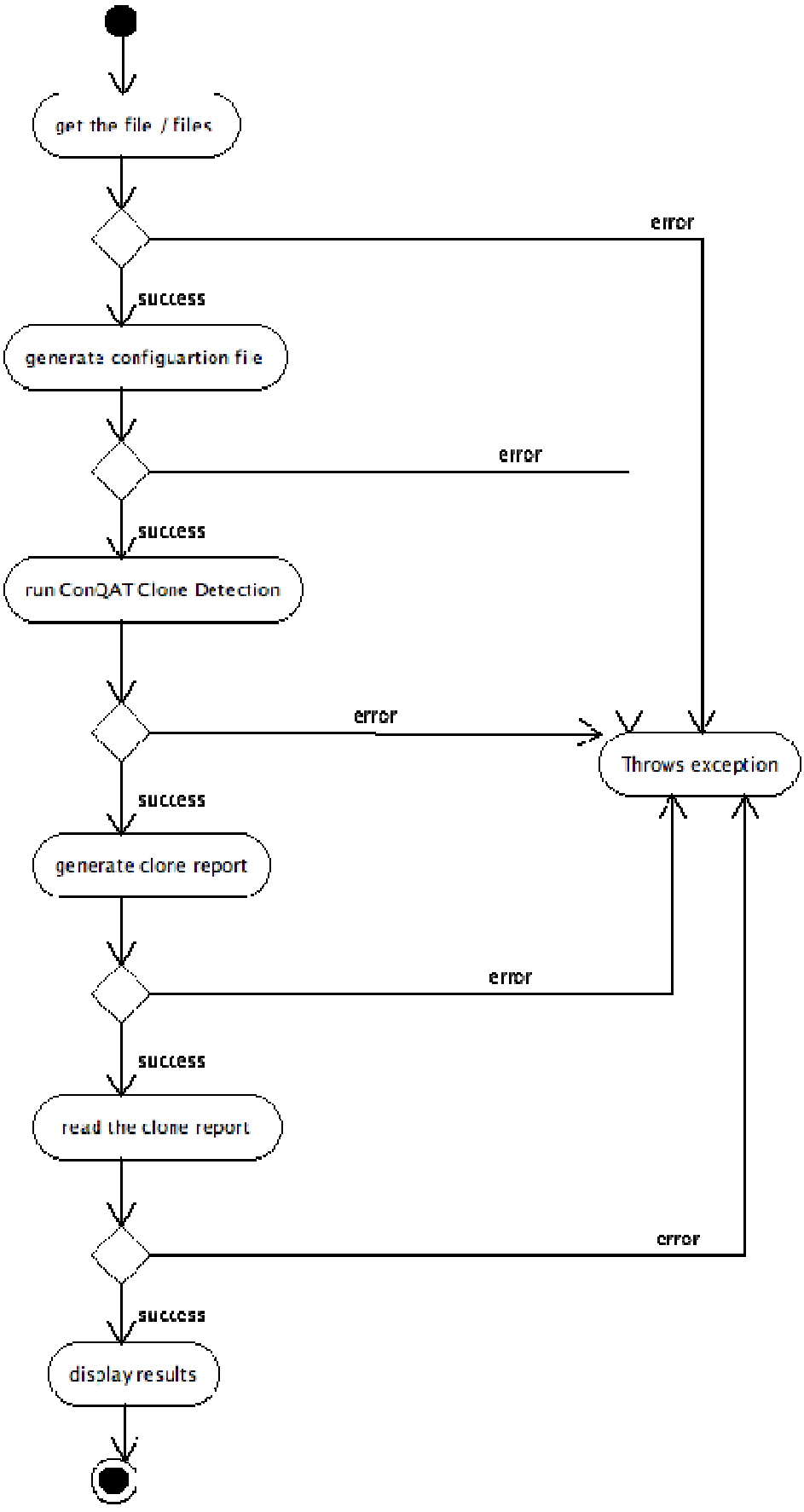


Figure 5.5 Activity diagram of Clone Detection

5.2.4 Preference Page

The usage of preference page allows user to modify the threshold values of checkstyle rules and to dis/en-able the analyzers. Up to now, the preference page is mainly for PL/I analyzers. The preference page can be modified for other languages analyzers in the future work. The working process of preference page with PL/I Checkstyle can be shown in the figure 5.6, which is activity diagram.

In the very beginning, the analyzers author has to define their rule checking criteria and the threshold value which justifies a circumstance that violates the rule criteria. When the Plug-in is launched, the criteria and the threshold values will be loaded. This is the initial preference value.

After loading this initial preference value, the value is stored into preference store, which is an extension point provided by Eclipse platform. After saving the values in the preference store, the values can be achieved whenever it is being called.

Before applying the preference values, user can modify the preference values whenever they want. When the preference values are modified, the new values are stored into the preference store. When the PL/I Checkstyle is started, the latest preference values are always used.

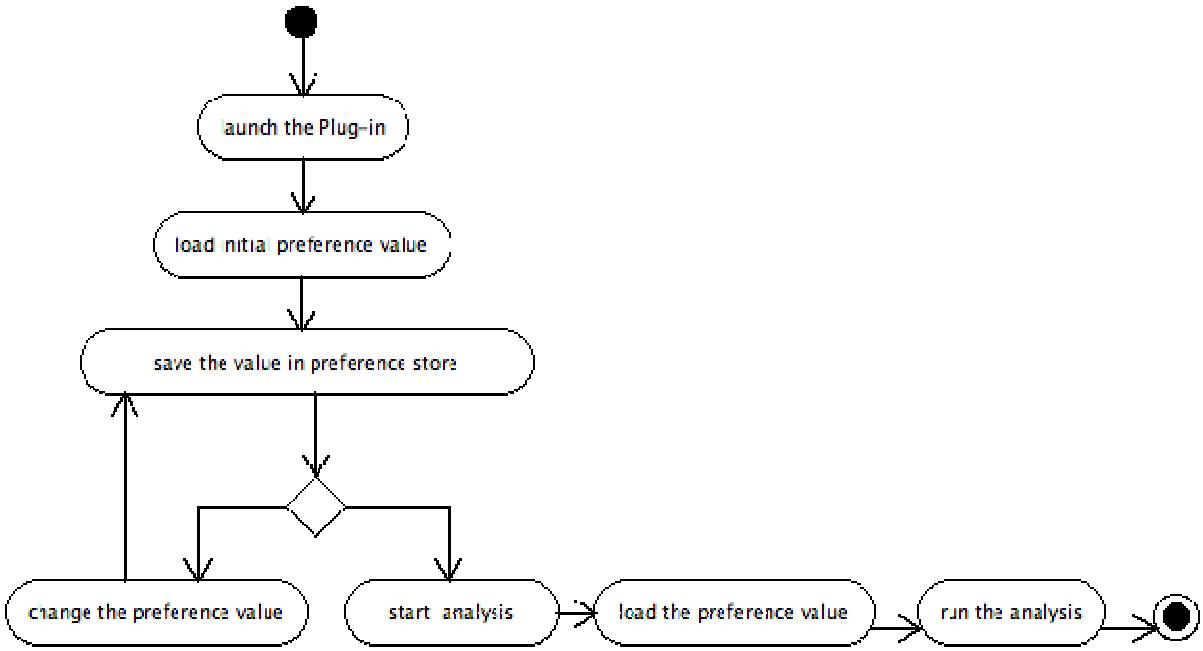


Figure 5.6 Activity diagram of Preference page

5.3 Legacy Checkstyle Platform

The Checkstyle platform is an Eclipse Plug-in that is composed by several packages. Each of the packages has different functionalities. The platform extends a couple of extension point of Eclipse, for example marker, view and preference etc. The platform provides an extension point for analyzers that do the code analysis, starts the analysis and displays the results. In figure 5.7, the package diagram of the Checkstyle platform.

Package: checkstyle.loading
This package contains the Plug-in activator. It starts the Plug-in and loads the views and preference page, which are going to display the results. It has a class to define the tools for the platform. Also, this package contains classes to activate customized functionalities of LpexEditor.

Package: checkstyle.tools
This package contains the starter of PL/I Checkstyle, Procedure Analysis and Clone detection. There are classes, which trigger off the tools. And this package has the classes to collect the analyzers from the Analyzers Plug-in. Also, it contains the classes that make the tools running in background- multi-threading mechanism.

Package: checkstyle.output
This package mainly contains output objects – `AnalyzerMessage` and procedure analysis XML report generator. They are the data models that provide for the presentation views.

Package: checkstyle.display.marker
This package contains the classes that extend marker as an extension of Eclipse markers. It makes the results presentation views much more interactive with the code editor.

Package: checkstyle.preference
This package contains the classes that extend preference page as an extension of Eclipse preference. It includes the graphics user interface implementation and preference value storage. Also, there is a class providing a public method to load these preference values.

Package: checkstyle.view
This package contains the implementation of all views – PL/I Checkstyle Problem and Info views, Clone Detection tree view and Procedure Call tree view. Apart from the graphic user interface implementation, it also contains the content provider and label provider classes.

Package : edu.tum.cs.conqat.clonedetective
This package contains the main algorithm of clone detection. And all the utilities that are used by the clone detection are also packed inside.

Package : edu.tum.cs.scanner
This package contains different programming language token scanner. This provides tools for extracting the token and getting the information from each of the tokens.

Package: checkstyle.utils
This package contains a lot of utilities for the other packages to call. For example, PL/I token generator and error handler.

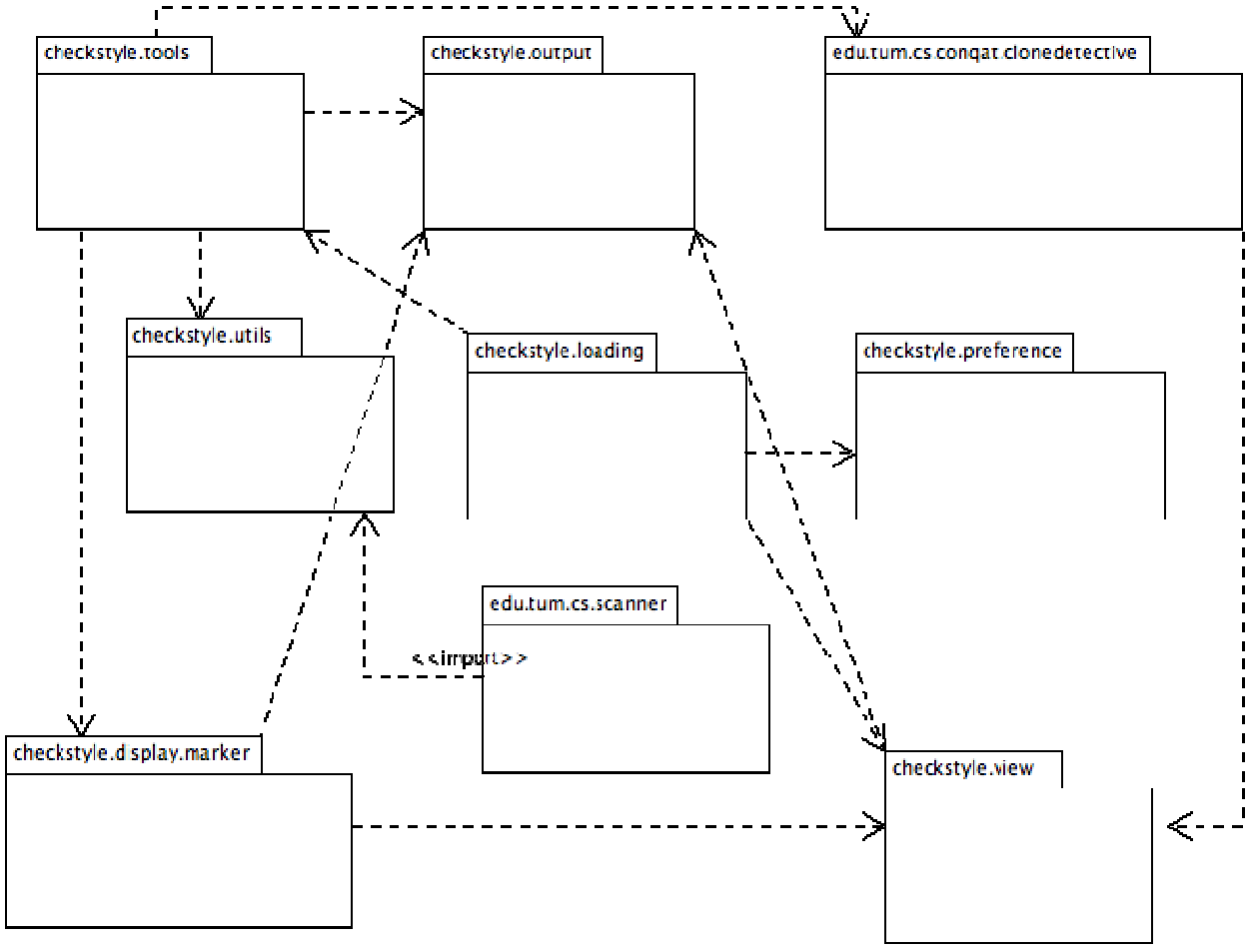


Figure 5.7 Package diagram of Legacy Checkstyle Platform

5.4 Analyzers

Analyzers can be regarded as another Plug-in package. As the meaning of the name, it contains all the analyzers that implement all the rules checking, which have already been discussed in chapter 4. In this Plug-in, the structure is rather simple. It has only three packages, as shown in figure 5.8.

Package: plicheckstyle.analyzers
This package mainly contains the analyzer classes, like LOCAnalyzer, LoopDepthAnalyzer, CommentRatioAnalyzer etc. These analyzers are responsible for analyzing the PL/I source code.

Package: plicheckstyle.analyzers.blockanalyzers
This package also contains some analyzer classes, but these classes are more than the analyzers in the above package. Their functionality is more or less similar to a lightweight parser. They can extract the block structure and analyze them. They provide parsing information for the analyzers.

Package: plicheckstyle.analyzers.utils
This package contains all tools, which might be called by any classes from the other packages. For example, it contains some objects, which store the lightweight parser information.

Normally, each of the packages in this Plug-in is possible to have dependency with the package in Checkstyle platform, especially the utilities package and the output package.

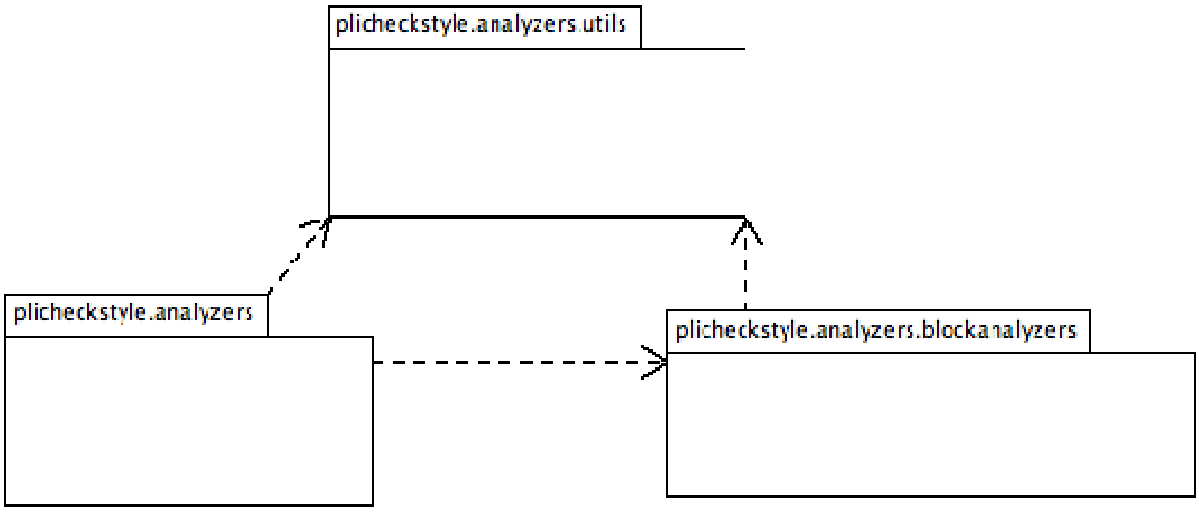


Figure 5.8 Analyzers Plug-in package diagram

5.5 Extending Analyzers

This section is about how to add more analyzers to the PL/I Checkstyle application. Since the Checkstyle platform provides an analyzer extension point, the analyzers extension project can be regarded as an Eclipse Plug-in project. Therefore, a plugin XML file is needed. The plugin.xml is shown as follows.

5.5.1 Creating the Plug-in xml file

```
<?xml version="1.0" encoding="UTF-8">
<plugin>
    <extension
        point="com.bmw.checkstyle.analyzer">
      <analyzer
       id="my.new.analyzer.locanalyzer"
       name="my first Lines of Code analyzer"
       class="my.new.analyzer.LOCAnalyzer"
       warning_criteria="LOC not larger than threshold value"
       threshold_value="100"
      />

    <!--
     add more tag <analyzer/> here
    -->

    </extension>
</plugin>
```

Figure 5.9 plugin.xml example

The extension element (in our plugin.xml file) describes an extension on the"`com.bmw.checkstyle.analyzer`" extension point. And we are extending the "`analyzer`". The `id` of the extension is "`my.new.analyzer.LOCAnalyzer`", which allows the Checkstyle platform to refer this particular analyzer. The `name` is "`my first Lines of Code analyzer`", which can be the description of the analyzer. Each analyzer has a class attribute and this specifies the class that implements the required interface – `IAnalyzer`. We have to implement this interface, so that the analyzer can provide functionality to analyze the code. The `warning_criteria` is a statement that states the warning criteria of analysis. When a situation that violates the criteria happens, the warning message will be sent to display. The `threshold_value` is the value that justifies if the `warning_criteria` has been violated.

In this case, an analyzer that checks lines of code of a file is going to be created. And if the line of code (LOC) in a file is more than 100, the warning message will show.

5.5.2 Implementing the IAnalyzer Interface

In this example, we are going to implement the interface – `IAnalyzer`. The interface is shown as follows:

```
public interface IAnalyzer{
  public List<AnalyzerMessage>  doAnalysis(IFile file) throws AnalyzerException ;
}
```

Figure 5.10 `IAnalyzer` interface

Therefore, `LOCAnalyzer` class has to be created and implements the method of `doAnalysis`. The class would be implemented as follows:

```
package my.new.analyzer;

import… // not display in detailed

public class LOCAnalyzer implements IAnalyzer{
    private int warning;
    private boolean enable;
    private final static String ANALYZER_ID=”my.new.analyzer.locanalyzer”
    private List<AnalyzerMessage> output = new ArrayList<AnalyzerMessage>

    public List<AnalyzerMessage> doAnalysis(IFile file){

      //step1: get the preference threshold value
       PreferenceLoader load = new PreferenceLoader();
       this.enable = load.getEnable(ANALYZER_ID);
       this.warning = load.getWarningThreshold(ANALYZER_ID);

      //step2: get the lines of code by Lpex Editor
       LpexView view = new LpexView(file.getLocation().toSting());
       int LOC = view.elements();


     // step 3: check if LOC is more than preference limit
      if(this.enable==true)&&(LOC > this.warning){
         String warn_message = “Too Many LOC”;
         AnalyzerMessage analyzermessage = new AnalyzerMessage(
AnalyzerMessage.MessageType.WARNING, warn_message, “-”, “”+LOC,1);
         this.output.add(analyzermessage);
      }


     // generate the info view results
       String info_message = “LOC”;
         AnalyzerMessage analyzermessage = new AnalyzerMessage(
       AnalyzerMessage.MessageType.INFO, info_message, “-”, “”+LOC,1);
       this.output.add(analyzermessage);

       return this.output;
    }

}
```

Figure 5.11 `LOCAnalyzer.java`

The `AnalyzerMessage` is composed of several fields: `MessageType` (type of the message, either WARNING or INFO), `entity` (the entity of the problem), `lineNumber` (line location of the problem, `message` (warning or info message), `value` (the value that measures the problem), `threshold` (threshold value defined in preference).

The `LOCAnalyzer.java` code checks the lines of code in a file. Firstly, it has to get the threshold value that defines the warning criteria. Secondly, it gets the LOC by using `LpexView` class, which is provided by Lpex Editor. In the code, `view.elements()` returns the LOC. Thirdly, by checking if the LOC is larger than the threshold value, the problem of the code can be found. Finally, generate the information to display as property information of the code analyzed.

Chapter 6

Prototype of PL\I Checkstyle Application

In this chapter, the prototype of PL/I Checkstyle Application is presented. The features of the application are partially discussed in the previous chapters. And the overall features are discussed here. The performance and accuracy of the checking will also be discussed in the following sections.

6.1 Overall Features Introduction

The features of the PL/I Checkstyle Application can be categorized as three parts: Procedure Calling Analysis, PL/I Analyzers and Clone Detection. Roughly speaking, they are doing the task of "PL/I Checkstyle"; however, due to their differences of data structure, presentation way and implementation, the features can be divided into 3 categories – Procedure Calling Analysis, PL/I Analyzers and Clone Detection. In figure 6.1, the features for the application can be structured in a map.

Procedure Calling Analysis provides information of calling relation between procedures within a file or files in a directory. Its output XML file is allowed to be accessed by other analyzers. The Procedure Calling Analyzer result is presented in a tree view.

PL/I Analyzers Plug-in is the core analysis part. It has already been discussed in the previous chapter. This part has extensible features. Users can extend its customized analyzers for PL/I or different programming languages. By using Plug-in mechanism, customized analyzers can be added or removed. Also, the rules checking criteria can be modified in the preference page feature. The analyzers results are shown in the problem view and info view.

Clone Detection provides information of duplicated code within files in directory. The output of the clone detection is a clone report. It can be accessed by clone report reader and displayed in a clone tree view.

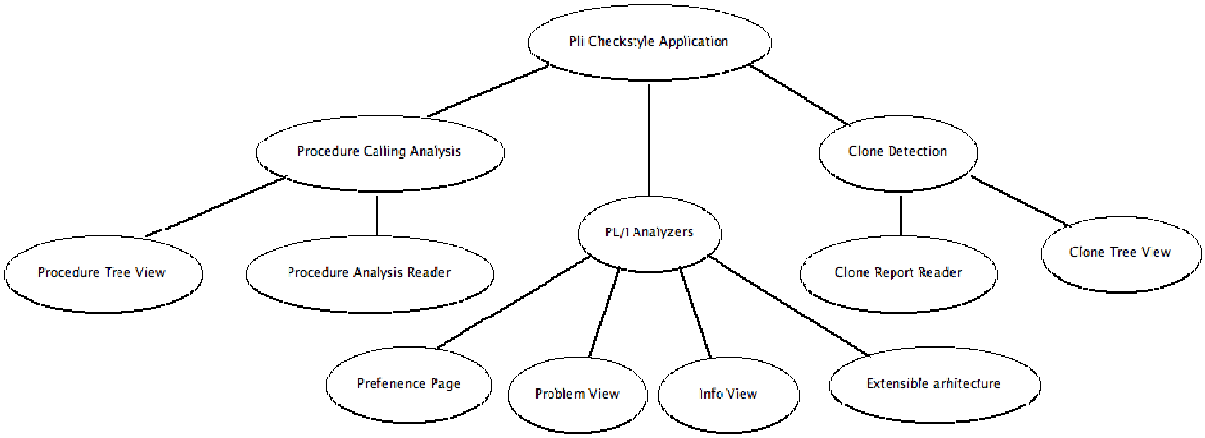


Figure 6.1 Overall Features Map

6.2 Prototype Graphic User Interface

The following figures are the graphic user interface of the prototype.

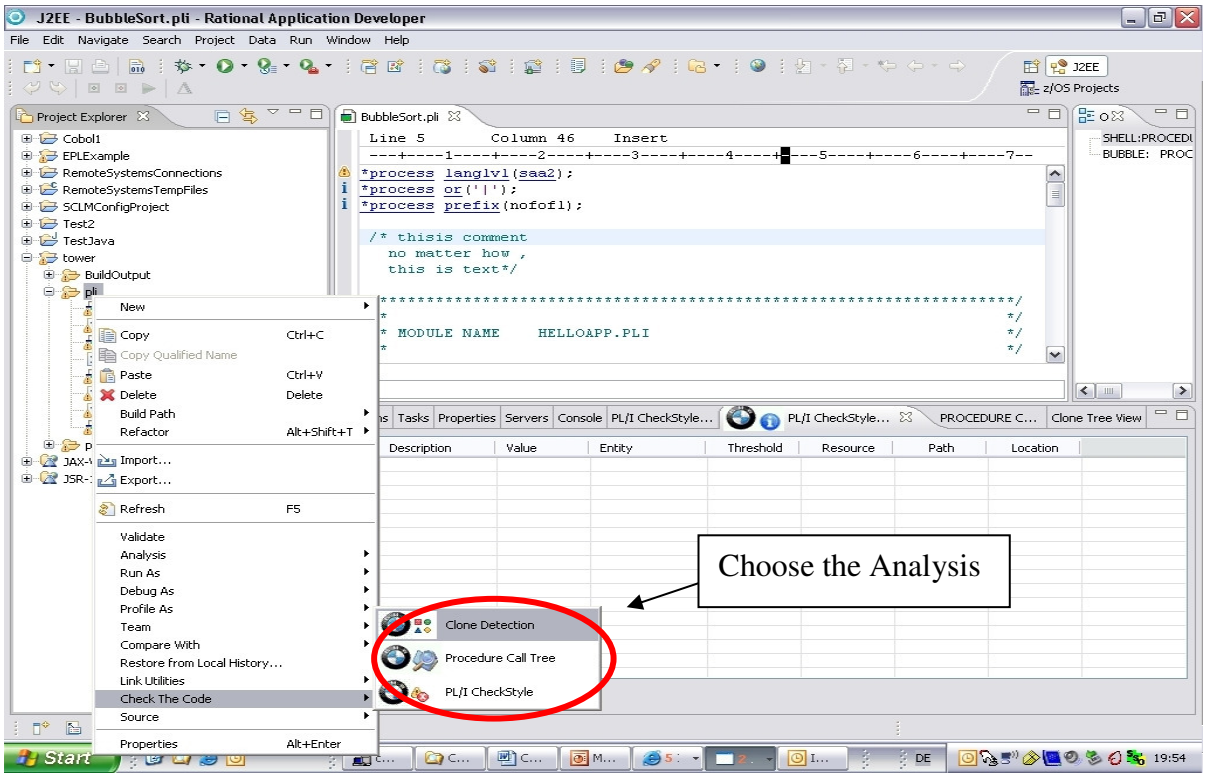


Figure 6.2 Choose the Analysis

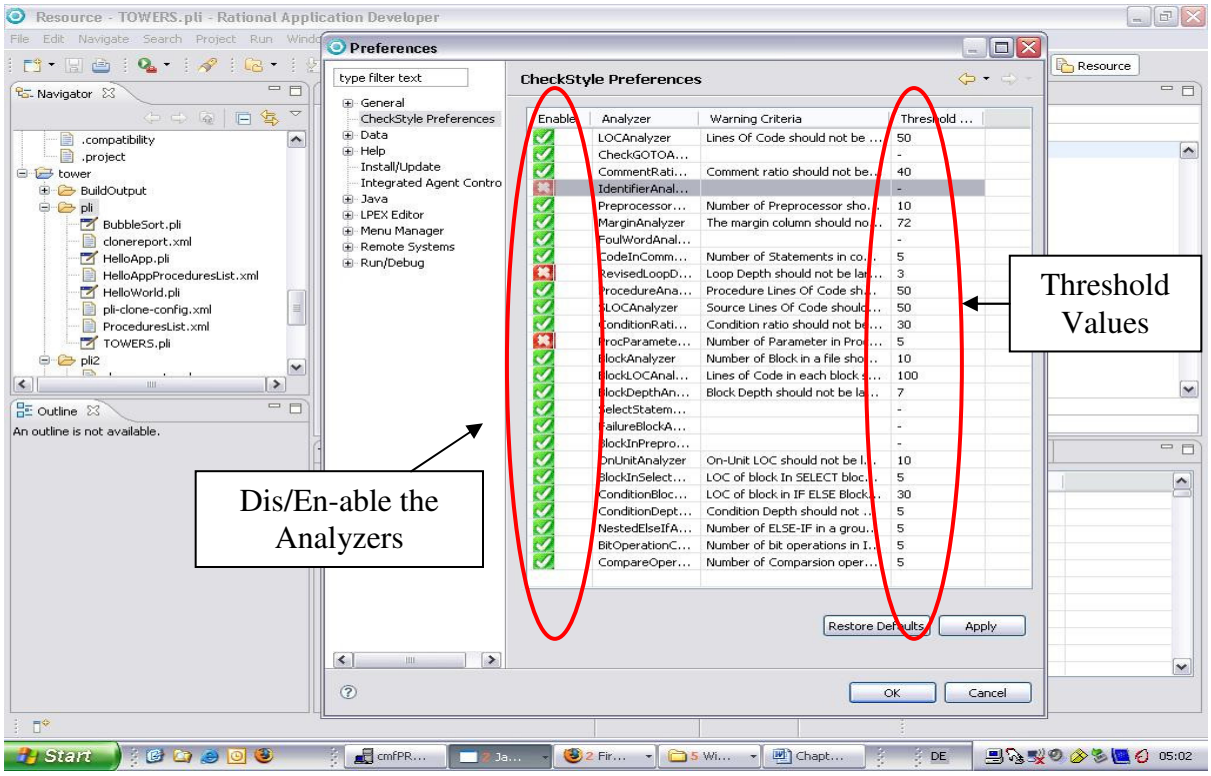


Figure 6.3 Preference Page for customizing the analyzers

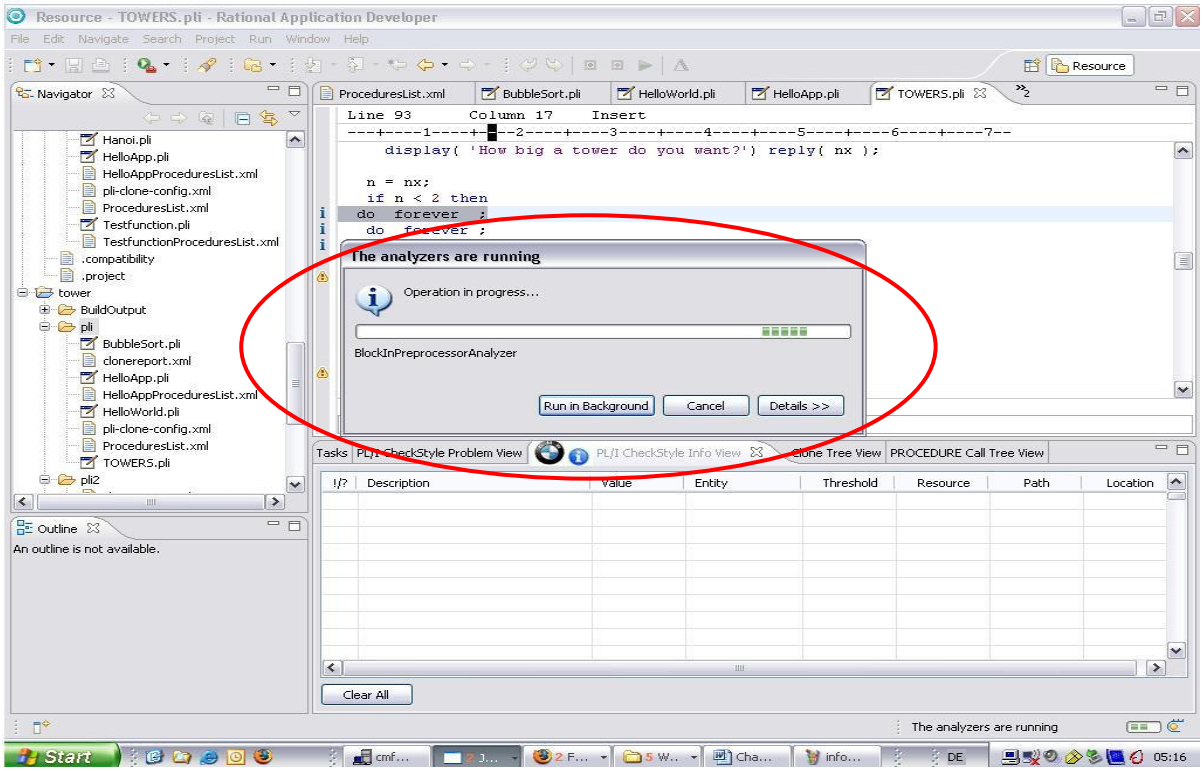


Figure 6.4 The analyzers running in a background

In Figure 6.4, shows that the application can allow the analyzers running parallel with the editor, so that it would not disturb the user when they trigger off PL/I Checkstyle. It also happens in Clone Detection and Procedure Call Analysis.

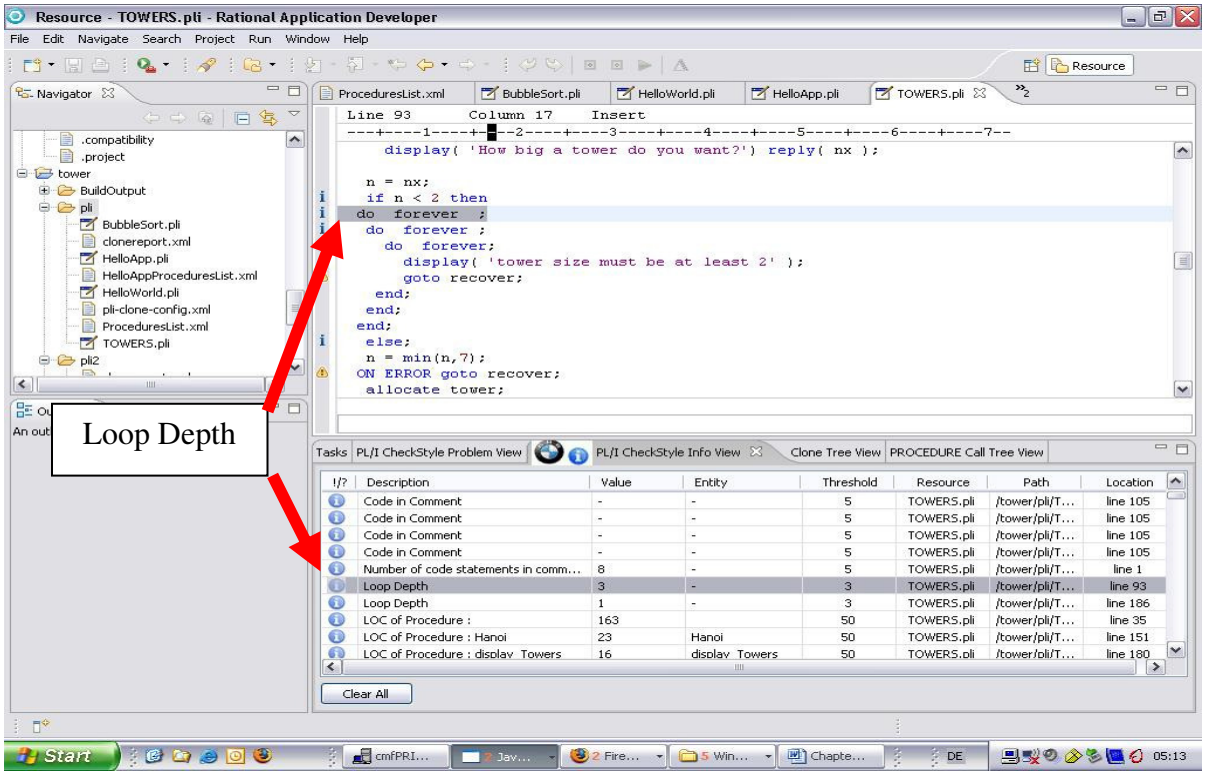


Figure 6.5 Info View

This view shows the properties information of the analyzed source code. By clicking the list, it can find the location of the information.

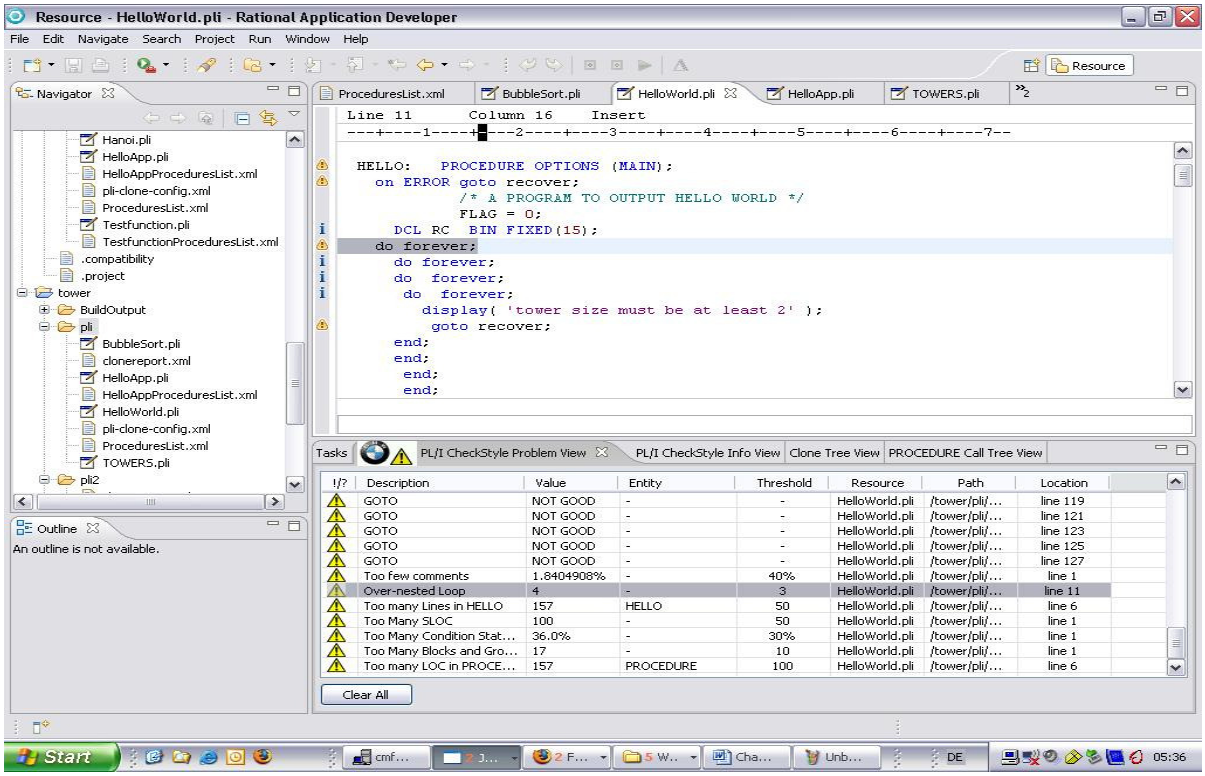


Figure 6.6 Problem View

The problem view only shows the analyzers results that violate the pre-defined criteria. As shown in figure 6.7, it shows that there is over-nested loop problem, which has 4 nested loops. To compare with the threshold value, which is 3, this is the reason why this appears in the problem view.

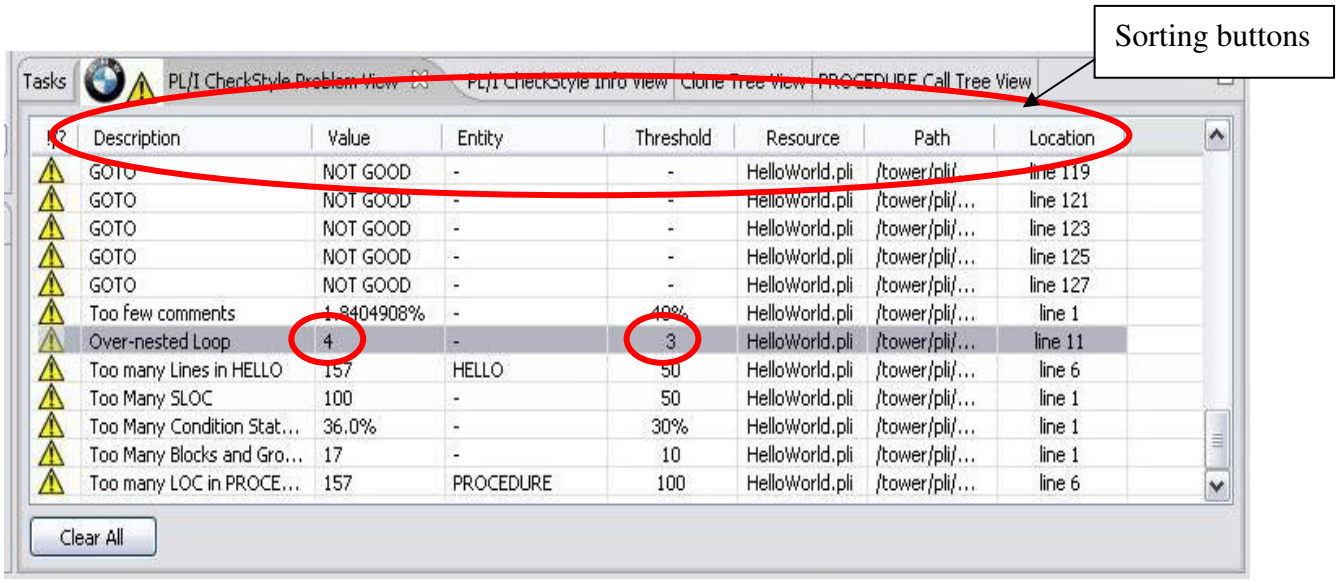


Figure 6.7 Results in Problem View

In the Info and Problem views, there are sorting buttons on the top of the table lists, which can help users to find the specific rules or location.

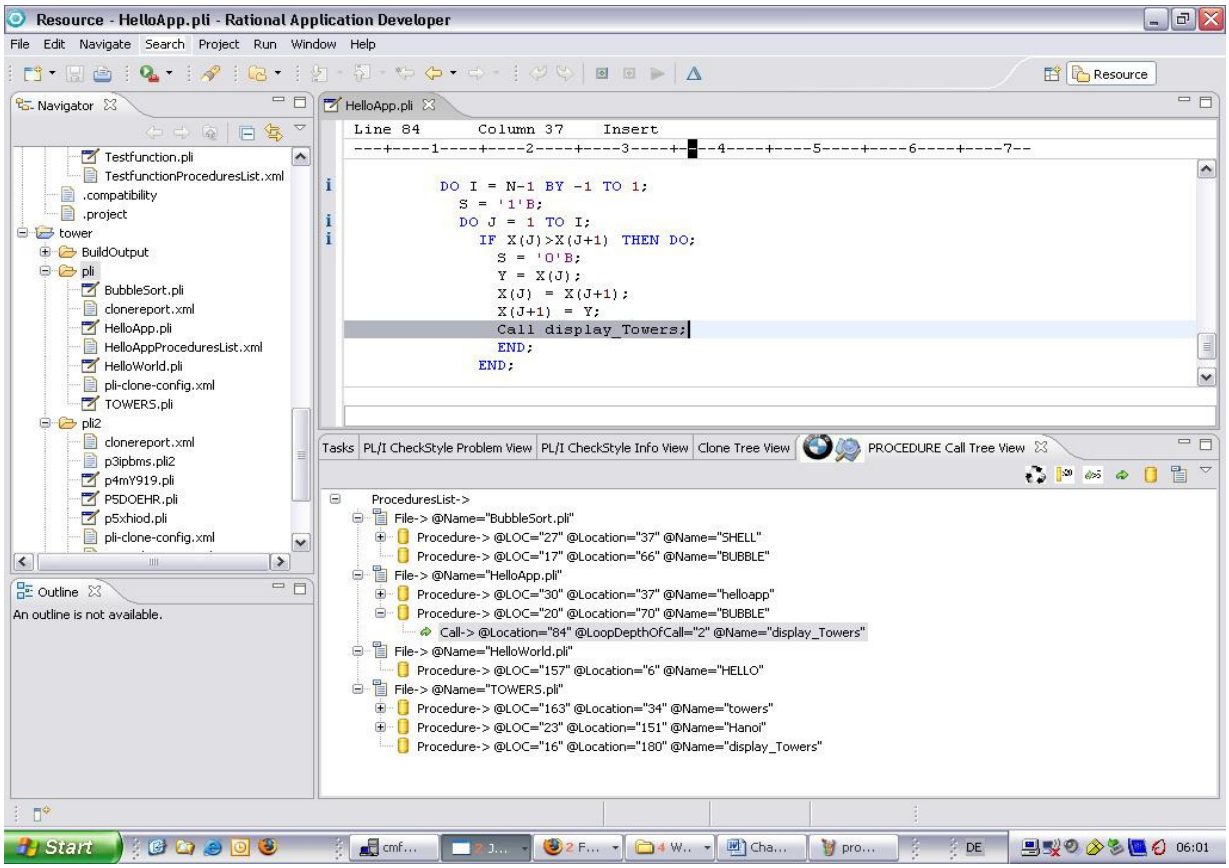


Figure 6.8 Procedure Call Tree View

The Procedure Call Analysis gives a tree view. It also has “Jump to Line” clicking, which locates the procedures or call statements in editor. Also, it has filter functionality in this view, as shown in figure 6.9. Therefore, user can filter out the information, which they want.

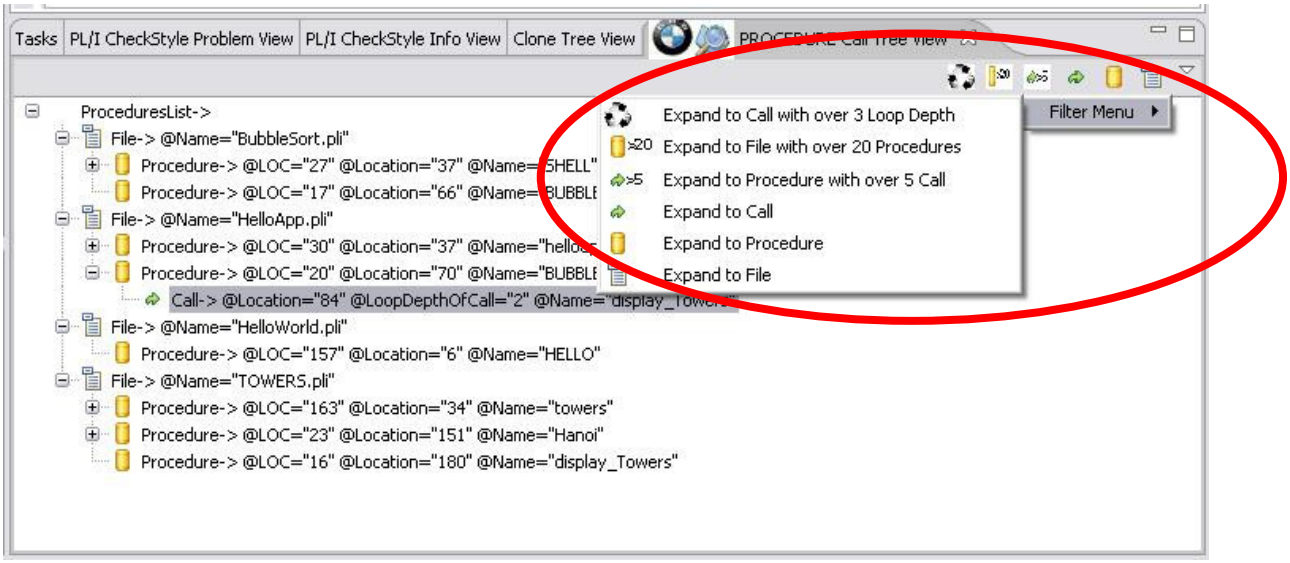


Figure 6.9 Filter Menus in Procedure Call Tree View

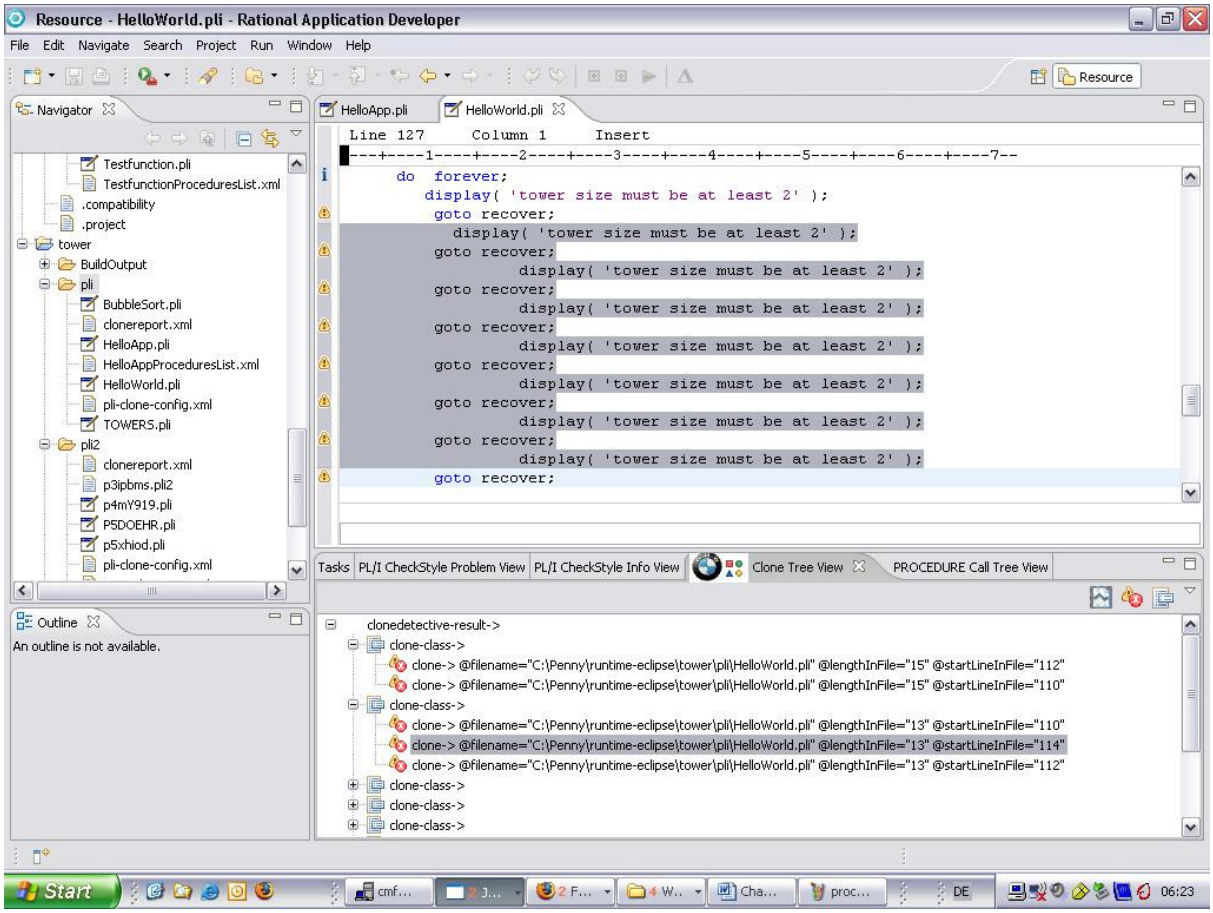


Figure 6.10 Clone Tree View

Clone Detection locates the duplicated codes within a directory. By clicking the clone in the view, it will jump and highlighted the codes that are cloned. Also, similar to Procedure Call View, Clone Tree View also provides filter menu for the user to filter out the clone information that they want.

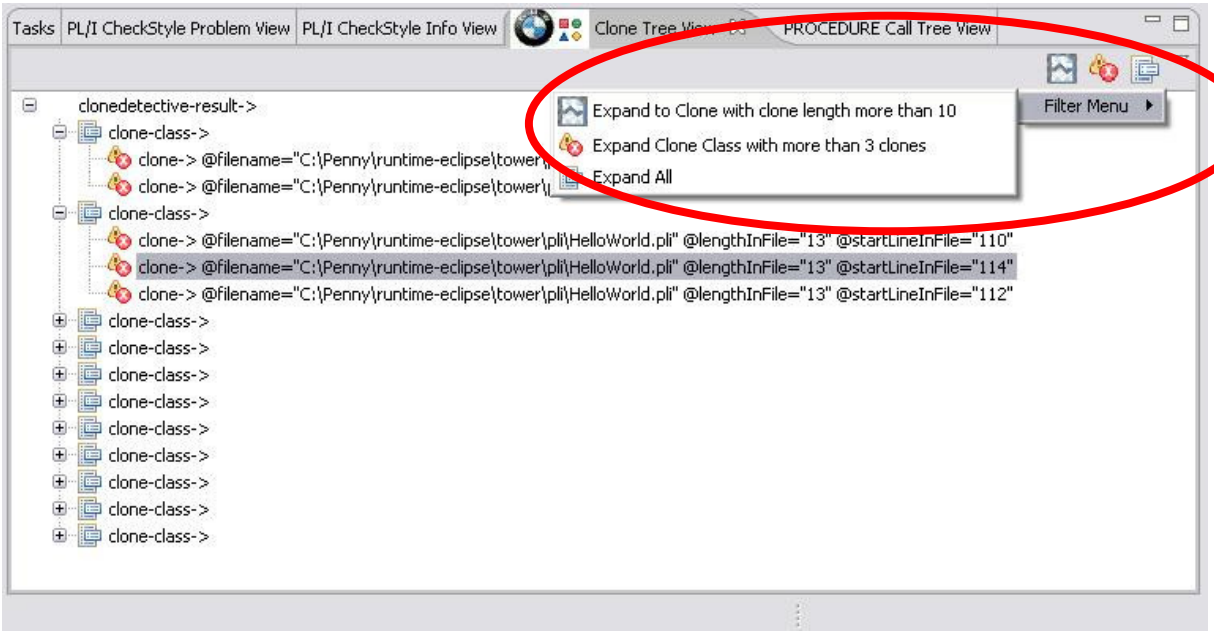


Figure 6.11 Filter Menus in Clone Tree View

6.3 Performance and Accuracy

In this section, we are going to discuss the performance and accuracy of the code analysis. The performance will be measured by the computational time. PL/I Checkstyle, Procedure Call Analysis and Clone Detection will be timed. For the accuracy, there are five test cases that will be conducted. Although it cannot be discussed the accuracy of all analyzers in here, 5 representative rules will be selected and tested.

6.3.1 Computational Time

For computational time issue, it depends on the time complexity of the analyzers. The following information is the computational time of the main analysis of the application.

6.3.1.1 PL/I Analyzers Computational time

The following list is the computational time (in milliseconds) of the analyzers. There are 26 analyzers in total. In order to measure the analysis time, a sample code with about 1000 LOC is used. The analyzers have to be executed several times, until it achieves the stable time results.

```
LOCAnalyzer time :265
CheckGOTOAnalyzer time :16
CommentRatioAnalyzer time :3
IdentifierAnalyzer time :937
PreprocessorAnalyzer time :250
MarginAnalyzer time :359
FoulWordAnalyzer time :125
CodeInCommentAnalyzer time :47
RevisedLoopDepthAnalyzer time :344
ProcedureAnalyzer time :672
SLOCAnalyzer time :47
ConditionRatioAnalyzer time :15
ProcParameterAnalyzer time :282
BlockAnalyzer time :1453
BlockLOCAnalyzer time :1703
BlockDepthAnalyzer time :1937
SelectStatementAnalyzer time :203
FailureBlockAnalyzer time :63
BlockInPreprocessorAnalyzer time :62
OnUnitAnalyzer time :63
BlockInSelectAnalyzer time :1297
ConditionBlockAnalyzer time :2453
ConditionDepthAnalyzer time :2813
NestedElseIfAnalyzer time :156
BitOperationComplexityAnalyzer time :328
CompareOperationComplexityAnalyzer time :2078
```

Total time = 17971 (~ 18 seconds)

6.3.1.2 Procedure Calling Analysis Computational time

Sample code with 1000 LOC is used.

Procedure Call relation time: 922 (~0.9 second)

6.3.1.3 Clone Detection Computational time

In this case, since the clone detection is comparing the all the files within a folder, it cannot only use one sample code to test. Therefore, a folder that are from BMW PL/I projects is used as a test sample. It contains 31 PL/I source files and average size of the files is 1000 LOC.

Clone Detection time: 4969 (~5 seconds)

6.3.1.4 Overall Performance

All in all, the overall performance is acceptable. The time complexity of the analyzers is normally O(n), i.e. linear. However, for those analyzers, like BlockAnalyzer and ConditionBlockAnalyzer, which do parsing, the time complexity of them is $O(n^2)$ or even $O(n^3)$. This is the reason why the computational time of those analyzers is significantly higher than the others. Also, the number of Plug-in in Eclipse will affect the performance too.

Also, the number of Plug-in in Eclipse would affect computational time. The more Plug-in in the Eclipse, the analysis would take more computational time. It is because it consumes more CPU workload, which may minimize the performance of the analyzers.

6.3.2 PL/I Checkstyle Accuracy

In order to measure the PL/I Checkstyle application accuracy, 5 analyzers are selected to be tested. Although it may not provide concrete evidence to prove the overall accuracy, 5 selected analyzers would be the representative analyzers in this application. It could give an idea of how accurate of the analysis. The following analyzers are selected to be tested.

- LoopDepthAnalyzer
- ProcParameterAnalyzer
- ConditionDepthAnalyzer
- NestedElseIfAnalyzer
- SelectStatementAnalyzer

Clone Detection and Procedure Calling Relation Analysis will not be discussed in here. It is because they are not in the main analyzers. Clone Detection is originally from ConQAT. ConQAT team has strongly proved the accuracy and stability of the Clone Detection. Therefore, it is not necessary to discuss here. For Procedure Calling Relation Analysis, since it is not a rule or standards to check, it only provides information for the analyzers, so in this section, it will not be discussed in here.

6.3.2.1 Test Case: LoopDepthAnalyzer

Loop mechanism is often used in a program. Therefore, nested looping group should be detected and analyzed. LoopDepthAnalyzer is responsible for checking the nesting level of nested looping group. If the number of nesting in a looping group is more than the threshold value, it will send a warning message to problem view.

Expected Results

The sample code A is shown in figure 6.12. There is 4-nested loop group. And the rule is that nested loop level is not allowed over 3. Therefore, the warning message "Over-nested loop" is expected to be in problem view. And the nesting loop level value should be in info view.

```
display_Towers: proc;

   dcl jx            fixed bin(31);
   /*hello world */

   put skip list( '' );
   do forever;
      do while (sqlcode = 0 & x < 16);
         do ix = 1 to 10;
             do jx = 1 to hbound(tower,2);

                 put skip list(    '   ' || tower(1,jx)
                         || '   ' || tower(2,jx)
                         || '   ' || tower(3,jx) );
             end;
          end;
        end;
      end;

   put skip list(    '   ' || '_' || '   ' || '_' || '   ' || '_' );

    end display_Towers;
   /* this is not copy*/
end towers;
```

Figure 6.12 Sample Code A

Test Results

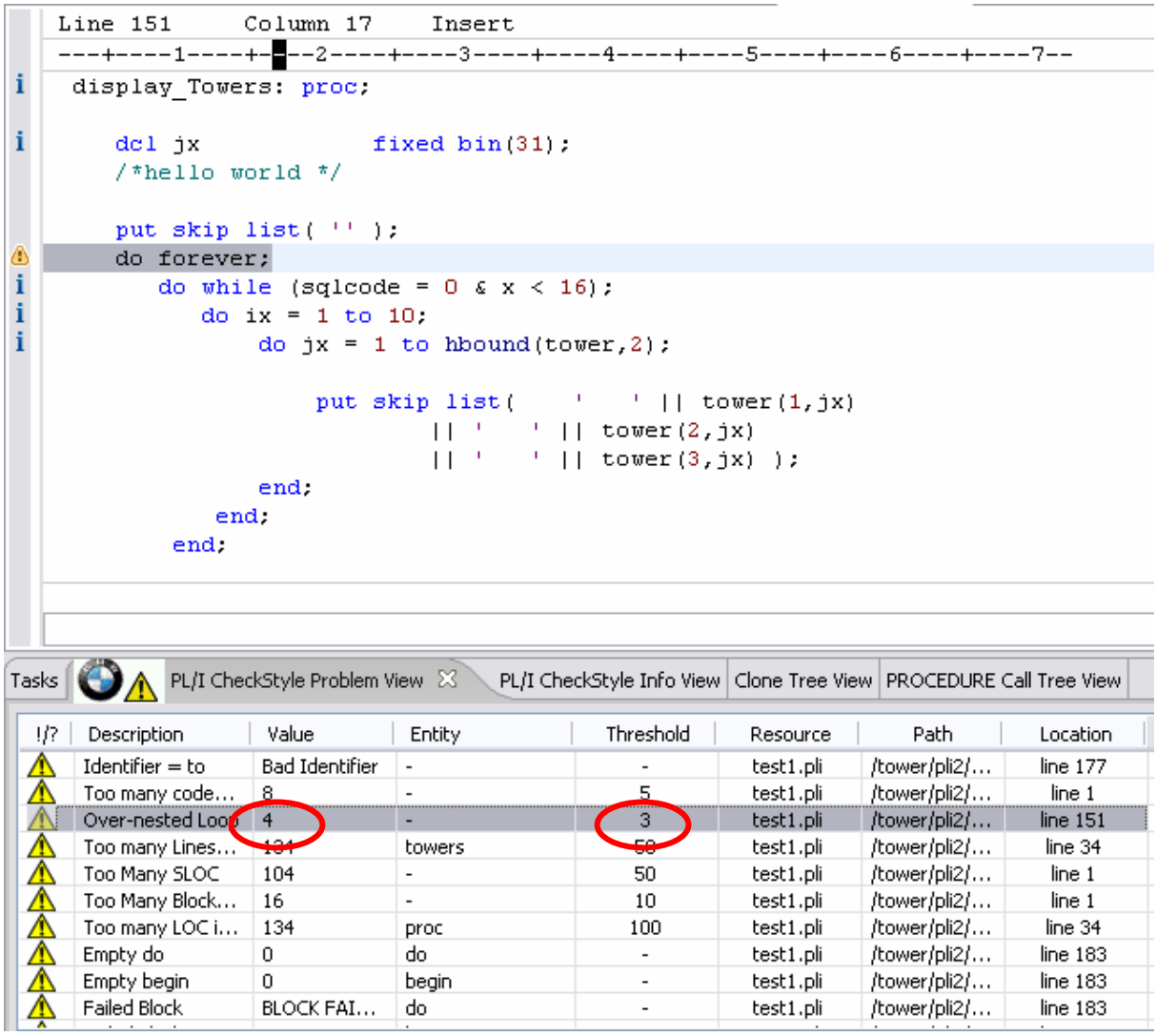


Figure 6.13 Test Result of Sample Code A in Problem View

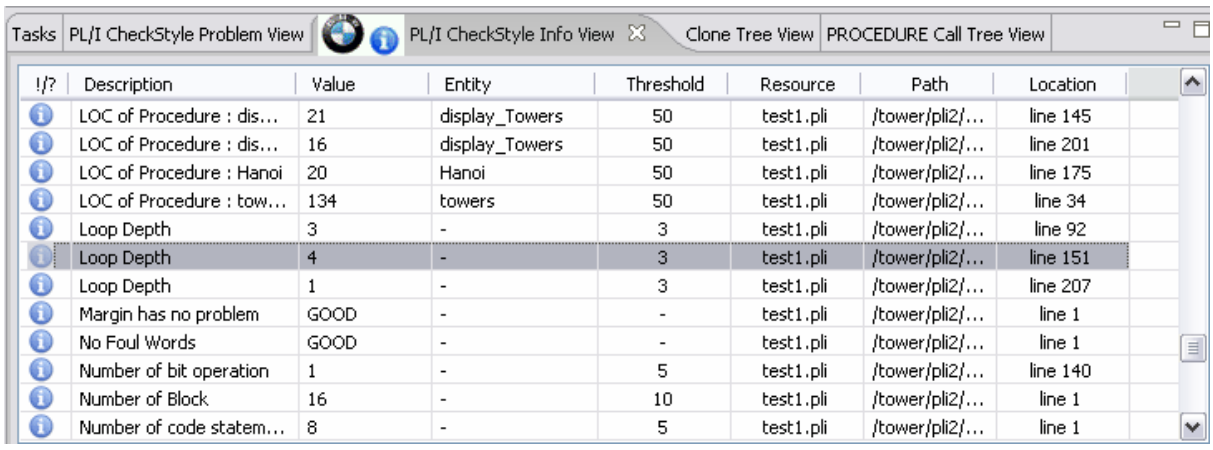


Figure 6.14 Test Result of Sample Code A in Info View

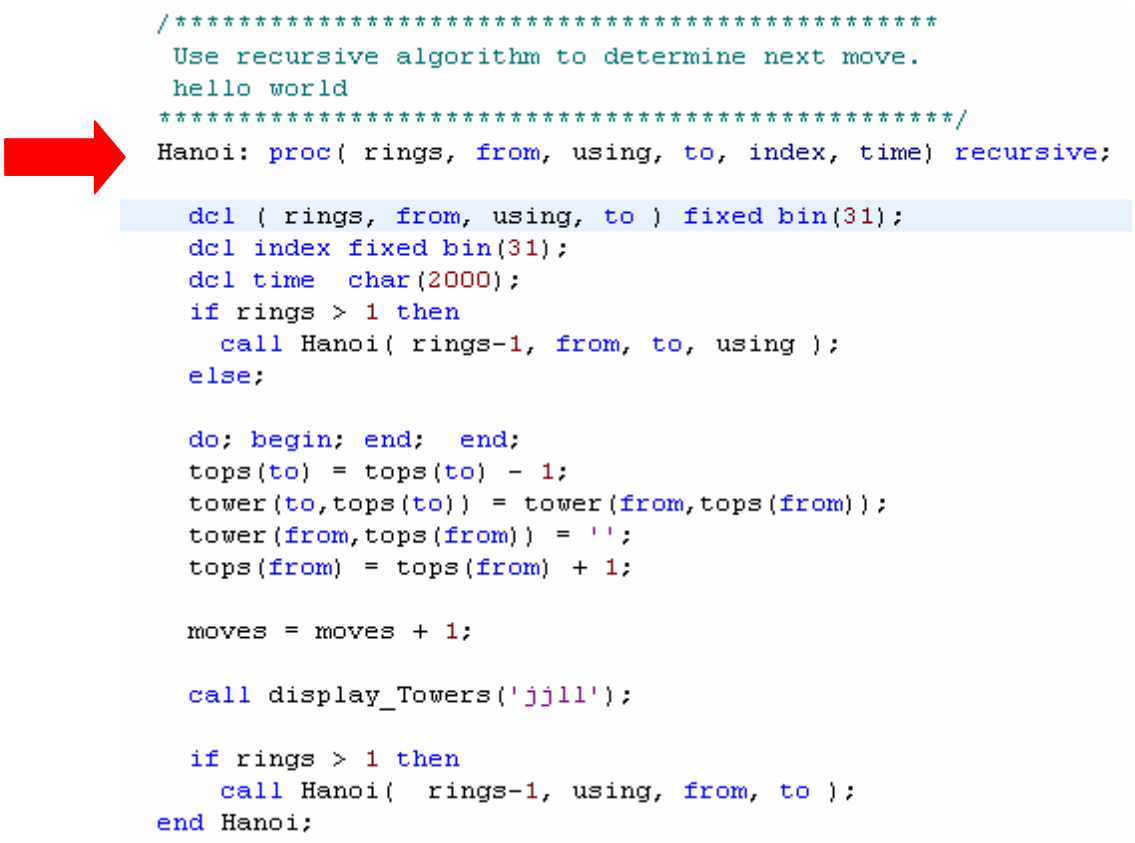
The test result is accurate. It shows both in Info and Problem view. It shows in Problem view because its value is 4, which is more than the threshold value 3.

6.3.2.2 Test Case: ProcParameterAnalyzer

The number of parameters passing in a procedure would affect the complexity of the procedure. And passing parameter in a procedure is very common in a programming. Therefore, the number of parameters passing in a procedure should be checked.

Expected Results

As shown in figure 6.15, there is a procedure called `Hanoi`. The procedure has 6 parameters passing, `rings, from, using, to, index, time`. And the rule is set that the number of parameter should not be more than 5. Therefore, there will be a warning message "Too many parameters" in the problem view. And this information will be shown in the info view.



```
/*************************************************
 Use recursive algorithm to determine next move.
 hello world
*************************************************/
Hanoi: proc( rings, from, using, to, index, time) recursive;

   dcl ( rings, from, using, to ) fixed bin(31);
   dcl index fixed bin(31);
   dcl time  char(2000);
   if rings > 1 then
     call Hanoi( rings-1, from, to, using );
   else;

   do; begin; end;  end;
   tops(to) = tops(to) - 1;
   tower(to,tops(to)) = tower(from,tops(from));
   tower(from,tops(from)) = '';
   tops(from) = tops(from) + 1;

   moves = moves + 1;

   call display_Towers('jjll');

   if rings > 1 then
     call Hanoi(  rings-1, using, from, to );
end Hanoi;
```

Figure 6.15 Sample Code B

Test Results

```
/*************************************************
 Use recursive algorithm to determine next move.
 hello world
*************************************************/
Hanoi: proc( rings, from, using, to, index, time) recursive;

  dcl ( rings, from, using, to ) fixed bin(31);
  dcl index fixed bin(31);
  dcl time  char(2000);
```

Tasks | PL/I CheckStyle Problem View | PL/I CheckStyle Info View | Clone Tree View | PROCEDURE Call Tree View

!/?	Description	Value	Entity	Threshold	Resource	Path	Location
⚠	Identifier = if	Bad Identifier	-	-	test1.pli	/tower/pli2/...	line 51
⚠	Identifier = index	Bad Identifier	-	-	test1.pli	/tower/pli2/...	line 178
⚠	Identifier = time	Bad Identifier	-	-	test1.pli	/tower/pli2/...	line 179
⚠	Identifier = to	Bad Identifier	-	-	test1.pli	/tower/pli2/...	line 52
⚠	Identifier = to	Bad Identifier	-	-	test1.pli	/tower/pli2/...	line 177
⚠	Over-nested Loop	4	-	3	test1.pli	/tower/pli2/...	line 151
⚠	Too few comments	26.52174%	-	40%	test1.pli	/tower/pli2/...	line 1
⚠	Too Many Blocks and Groups	16	-	10	test1.pli	/tower/pli2/...	line 1
⚠	Too many codes in Comment	8	-	5	test1.pli	/tower/pli2/...	line 1
⚠	Too many Lines in towers	134	towers	50	test1.pli	/tower/pli2/...	line 34
⚠	Too many LOC	230	-	50	test1.pli	/tower/pli2/...	line 1
⚠	Too many LOC in ON-Unit	12	ON-Unit	10	test1.pli	/tower/pli2/...	line 72
⚠	Too many LOC in proc	134	proc	100	test1.pli	/tower/pli2/...	line 34
⚠	Too many parameters	6	Hanoi	5	test1.pli	/tower/pli2/...	line 175
⚠	Too Many SLOC	196	-	50	test1.pli	/tower/pli2/...	line 1

Figure 6.16 Test Result of Sample Code B in Problem View

Tasks | PL/I CheckStyle Problem View | PL/I CheckStyle Info View | Clone Tree View | PROCEDURE Call Tree View

!/?	Description	Value	Entity	Threshold	Resource	Path	Location
ℹ	Number of comparison oper...	1	-	5	test1.pli	/tower/pli2/...	line 194
ℹ	Number of GOTO	5	-	-	test1.pli	/tower/pli2/...	line 1
ℹ	Number of parameters : 0	0	towers	5	test1.pli	/tower/pli2/...	line 34
ℹ	Number of parameters : 0	0	display_Hall	5	test1.pli	/tower/pli2/...	line 145
ℹ	Number of parameters : 0	0	display_Towers	5	test1.pli	/tower/pli2/...	line 202
ℹ	Number of parameters : 6	6	Hanoi	5	test1.pli	/tower/pli2/...	line 175
ℹ	ON_STATEMENT error	ON_STATE...	error	10	test1.pli	/tower/pli2/...	line 56
ℹ	ON_STATEMENT ERROR	ON_STATE...	ERROR	10	test1.pli	/tower/pli2/...	line 102
ℹ	ON_UNIT in Block : begin	ON_UNIT	begin	10	test1.pli	/tower/pli2/...	line 72
ℹ	Preprocessor Type	PREPROCE...	test1.pli	-	test1.pli	/tower/pli2/...	line 1
ℹ	Preprocessor Type	PREPROCE...	test1.pli	-	test1.pli	/tower/pli2/...	line 2
ℹ	Preprocessor Type	PREPROCE...	test1.pli	-	test1.pli	/tower/pli2/...	line 3
ℹ	proc Depth Level	1	proc	7	test1.pli	/tower/pli2/...	line 34
ℹ	proc Depth Level	2	proc	7	test1.pli	/tower/pli2/...	line 145

Figure 6.17 Test Result of Sample Code B in Info View

The test result is precise. The result shows in both problem view and info view, because of the number of parameters violating the rule. Also, the other procedures with zero parameter will not be shown in problem view.

6.3.2.3 Test Case: ConditionDepthAnalyzer

IF-statement is also very commonly used in a program. However, IF-statement logic sometimes is not easily to understand, especially for the nested IF-statement (i.e. IF-statement within another IF-statement). ConditionDepthAnalyzer checks the nesting level of IF-statement. Also, the number of nesting level of IF-statement should be checked.

Expected Results

In figure 18, there is a sample code C which has a nested IF-statement. And in the figure, it labels the nesting level of the IF-statements. The nesting level of IF-statement will be checked. The rule is set that IF-statement depth should not be larger than 2. Therefore, the IF-statements labeled as level 3 will have warning message "Too Deep IF statement Depth". And all IF-statements nesting level will be shown in Info view.

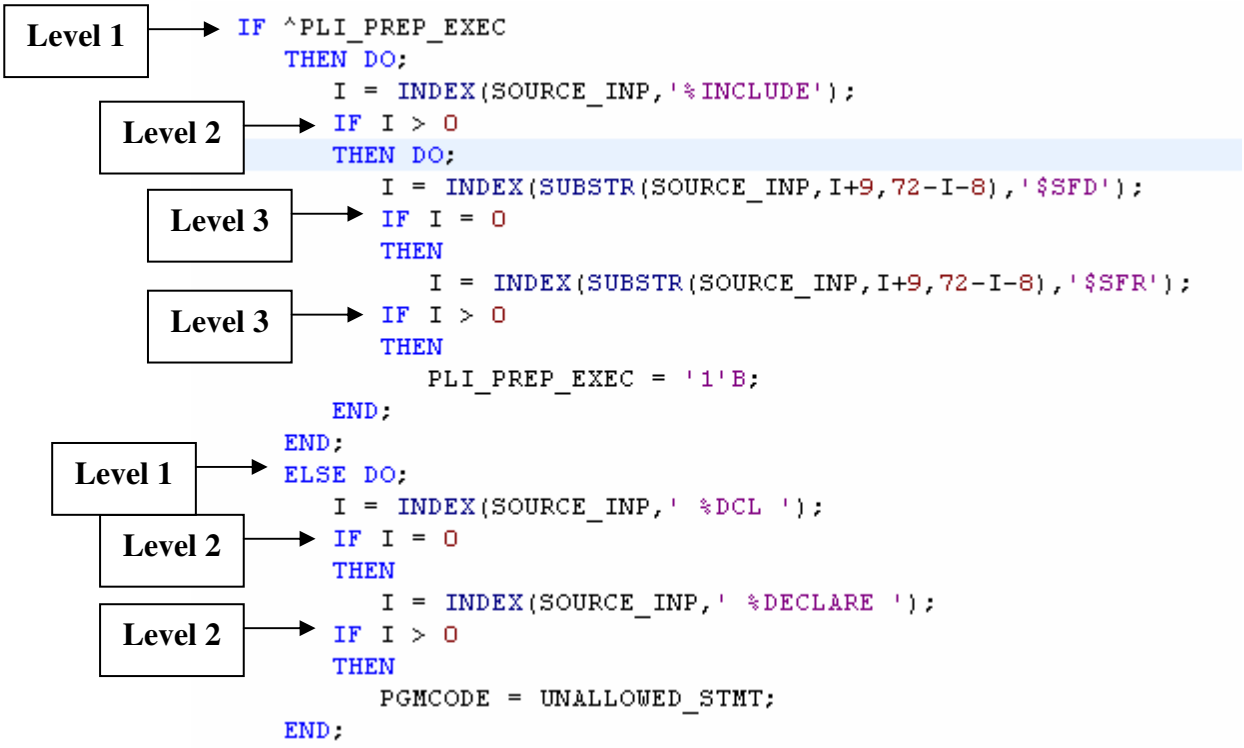


Figure 6.18 Sample Code C

Test Results

```
IF ^PLI_PREP_EXEC
   THEN DO;
      I = INDEX(SOURCE_INP,'%INCLUDE');
      IF I > 0
      THEN DO;
         I = INDEX(SUBSTR(SOURCE_INP,I+9,72-I-8),'$SFD');
         IF I = 0
         THEN
            I = INDEX(SUBSTR(SOURCE_INP,I+9,72-I-8),'$SFR');
         IF I > 0
         THEN
            PLI_PREP_EXEC = '1'B;
      END;
   END;
   ELSE DO;
      I = INDEX(SOURCE_INP,' %DCL ');
```

Tasks | PL/I CheckStyle Problem View | PL/I CheckStyle Info View | Clone Tree View | PROCEDURE Call Tree View

!/?	Description	Value	Entity	Threshold	Resource	Path	Location
⚠	GOTO	NOT GOOD	-	-	HelloWorld.pli	/tower/pli/...	line 121
⚠	GOTO	NOT GOOD	-	-	HelloWorld.pli	/tower/pli/...	line 123
⚠	GOTO	NOT GOOD	-	-	HelloWorld.pli	/tower/pli/...	line 125
⚠	GOTO	NOT GOOD	-	-	HelloWorld.pli	/tower/pli/...	line 127
⚠	Missing OTHERWISE case	NOT GOOD!	SELECT	-	HelloWorld.pli	/tower/pli/...	line 96
⚠	Over-nested Loop	4	-	3	HelloWorld.pli	/tower/pli/...	line 11
⚠	Too Deep IF statement Depth	Level 3	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 44
⚠	Too Deep IF statement Depth	Level 3	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 47
⚠	Too few comments	1.8404908%	-	40%	HelloWorld.pli	/tower/pli/...	line 1
⚠	Too Many Blocks and Groups	17	-	10	HelloWorld.pli	/tower/pli/...	line 1
⚠	Too Many Condition Statments	36.0%	-	30%	HelloWorld.pli	/tower/pli/...	line 1

Figure 6.19 Test Result of Sample Code C in Problem View

Tasks | PL/I CheckStyle Problem View | PL/I CheckStyle Info View | Clone Tree View | PROCEDURE Call Tree View

!/?	Description	Value	Entity	Threshold	Resource	Path	Location
ℹ	Identifier = RC	OK	-	-	HelloWorld.pli	/tower/pli/...	line 10
ℹ	IF statement Depth Level	Level 1	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 25
ℹ	IF statement Depth Level	Level 1	ELSE_UNIT	2	HelloWorld.pli	/tower/pli/...	line 27
ℹ	IF statement Depth Level	Level 1	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 38
ℹ	IF statement Depth Level	Level 2	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 41
ℹ	IF statement Depth Level	Level 3	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 44
ℹ	IF statement Depth Level	Level 3	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 47
ℹ	IF statement Depth Level	Level 1	ELSE_UNIT	2	HelloWorld.pli	/tower/pli/...	line 52
ℹ	IF statement Depth Level	Level 2	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 54
ℹ	IF statement Depth Level	Level 2	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 57
ℹ	IF statement Depth Level	Level 1	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 62
ℹ	IF statement Depth Level	Level 1	ELSEIF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 64
ℹ	IF statement Depth Level	Level 1	ELSEIF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 66
ℹ	IF statement Depth Level	Level 2	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 68
ℹ	IF statement Depth Level	Level 2	ELSEIF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 70
ℹ	IF statement Depth Level	Level 1	ELSEIF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 74

Figure 6.20 Test Result of Sample Code C in Info View

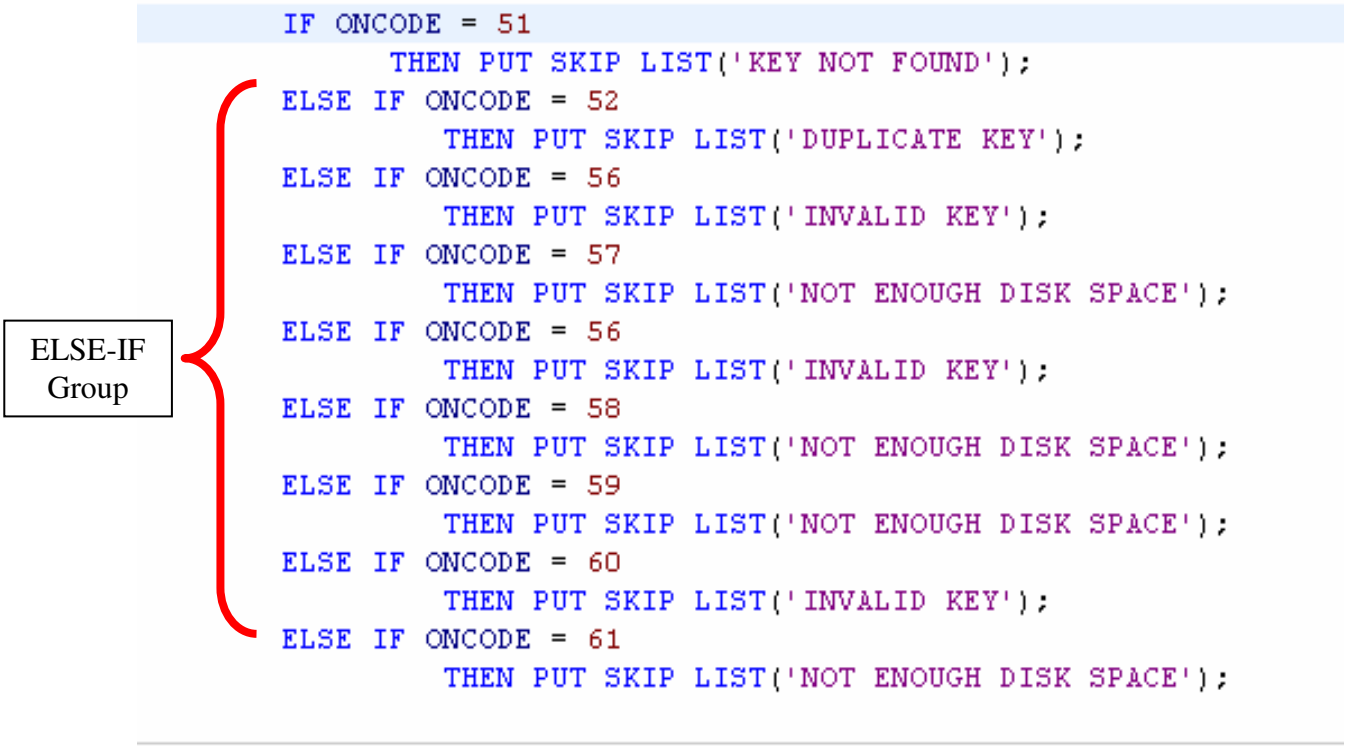
Test result is accurate. The nesting level of IF-statement with 3 is shown as a problem. Also, in the Info view, the indicated lists are the information of sample code C.

6.3.2.4 Test Case: NestedElseIfAnalyzer

NestedElseIfAnalyzer checks the IF-statement that has ELSE-IF logic. It has already been discussed in the previous chapter. It is actually same as SELECT group. Therefore the number of ELSE-IF in a group should be checked if it is more than a certain number, because SELECT group is much easier to be understood.

Expected Results

As shown in figure 6.21, a sample code with IF-statement that has a group of ELSE-IF logic. And the number of ELSE-IF in a group is more than the threshold value, which is set by 5. Therefore, based on this piece of code, there will be warning message "Too many ELSE-IF in a group" in problem view. Also, the number of ELSE-IF will also be shown in info view.




```
IF ONCODE = 51
      THEN PUT SKIP LIST('KEY NOT FOUND');
ELSE IF ONCODE = 52
      THEN PUT SKIP LIST('DUPLICATE KEY');
ELSE IF ONCODE = 56
      THEN PUT SKIP LIST('INVALID KEY');
ELSE IF ONCODE = 57
      THEN PUT SKIP LIST('NOT ENOUGH DISK SPACE');
ELSE IF ONCODE = 56
      THEN PUT SKIP LIST('INVALID KEY');
ELSE IF ONCODE = 58
      THEN PUT SKIP LIST('NOT ENOUGH DISK SPACE');
ELSE IF ONCODE = 59
      THEN PUT SKIP LIST('NOT ENOUGH DISK SPACE');
ELSE IF ONCODE = 60
      THEN PUT SKIP LIST('INVALID KEY');
ELSE IF ONCODE = 61
      THEN PUT SKIP LIST('NOT ENOUGH DISK SPACE');
```

Figure 6.21 Sample Code D

Test Results

```
        IF ONCODE = 51
              THEN PUT SKIP LIST('KEY NOT FOUND');
        ELSE IF ONCODE = 52
                  THEN PUT SKIP LIST('DUPLICATE KEY');
        ELSE IF ONCODE = 56
                  THEN PUT SKIP LIST('INVALID KEY');
        ELSE IF ONCODE = 57
                  THEN PUT SKIP LIST('NOT ENOUGH DISK SPACE');
        ELSE IF ONCODE = 56
                  THEN PUT SKIP LIST('INVALID KEY');
        ELSE IF ONCODE = 58
                  THEN PUT SKIP LIST('NOT ENOUGH DISK SPACE');
        ELSE IF ONCODE = 59
                  THEN PUT SKIP LIST('NOT ENOUGH DISK SPACE');
        ELSE IF ONCODE = 60
                  THEN PUT SKIP LIST('INVALID KEY');
        ELSE IF ONCODE = 61
                  THEN PUT SKIP LIST('NOT ENOUGH DISK SPACE');
```

Tasks | PL/I CheckStyle Problem View | PL/I CheckStyle Info View | Clone Tree View | PROCEDURE Call Tree View

!/?	Description	Value	Entity	Threshold	Resource	Path	Location
⚠	Too Deep IF statement Depth	Level 3	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 47
⚠	Too few comments	1.8404908%	-	40%	HelloWorld.pli	/tower/pli/...	line 1
⚠	Too Many Blocks and Groups	17	-	10	HelloWorld.pli	/tower/pli/...	line 1
⚠	Too Many Condition Statments	36.0%	-	30%	HelloWorld.pli	/tower/pli/...	line 1
⚠	Too many ELSE-IF in a group	8	ELSE-IF in Level...	5	HelloWorld.pli	/tower/pli/...	line 79
⚠	Too many Lines in HELLO	157	HELLO	50	HelloWorld.pli	/tower/pli/...	line 6
⚠	Too many LOC	163	-	50	HelloWorld.pli	/tower/pli/...	line 1
⚠	Too many LOC in PROCEDURE	157	PROCEDURE	100	HelloWorld.pli	/tower/pli/...	line 6
⚠	Too many LOC in WHEN or OTH...	16	SELECT	5	HelloWorld.pli	/tower/pli/...	line 111
⚠	Too Many SLOC	100	-	50	HelloWorld.pli	/tower/pli/...	line 1

Figure 6.22 Test Result of Sample Code D in Problem View

Tasks | PL/I CheckStyle Problem View | PL/I CheckStyle Info View | Clone Tree View | PROCEDURE Call Tree View

!/?	Description	Value	Entity	Threshold	Resource	Path	Location
ℹ	Number of comparison oper...	1	-	5	HelloWorld.pli	/tower/pli/...	line 89
ℹ	Number of comparison oper...	1	-	5	HelloWorld.pli	/tower/pli/...	line 91
ℹ	Number of comparison oper...	1	-	5	HelloWorld.pli	/tower/pli/...	line 93
ℹ	Number of ELSE-IF in a group	3	ELSE-IF in Level...	5	HelloWorld.pli	/tower/pli/...	line 64
ℹ	Number of ELSE-IF in a group	8	ELSE-IF in Level...	5	HelloWorld.pli	/tower/pli/...	line 79
ℹ	Number of ELSE-IF in a group	2	ELSE-IF in Level...	5	HelloWorld.pli	/tower/pli/...	line 70
ℹ	Number of GOTO	11	-	-	HelloWorld.pli	/tower/pli/...	line 1
ℹ	Number of parameters : 0	0	HELLO	5	HelloWorld.pli	/tower/pli/...	line 6
ℹ	ON_STATEMENT ERROR	ON_STATE...	ERROR	10	HelloWorld.pli	/tower/pli/...	line 7
ℹ	Preprocessor Type	PREPROCE...	HelloWorld.pli	-	HelloWorld.pli	/tower/pli/...	line 1
ℹ	Preprocessor Type	PREPROCE...	HelloWorld.pli	-	HelloWorld.pli	/tower/pli/...	line 2
ℹ	Preprocessor Type	PREPROCE...	HelloWorld.pli	-	HelloWorld.pli	/tower/pli/...	line 3
ℹ	PROCEDURE Depth Level	1	PROCEDURE	7	HelloWorld.pli	/tower/pli/...	line 6
ℹ	SELECT Depth Level	2	SELECT	7	HelloWorld.pli	/tower/pli/...	line 96

Figure 6.23 Test Result of Sample Code D in Info View

The result is correct. The number of ELSE-IF in a group is 8 in sample code D, which is more than the threshold value, so that problem view shows the warning message.

6.3.2.5 Test Case: SelectStatementAnalyzer

SelectStatementAnalyzer provides the analysis of counting the number of cases in a SELECT group and check if the default case is defined. A SELECT statement without default case is a potential bug in a program, because exceptional case may happen. Also, SELECT group is also commonly used in a program. Therefore, this analyzer is selected to test.

Expected Results

In the figure 6.24, there is a SELECT group with 13 cases, but without defining OTHERWISE case. Therefore, warning message “missing OTHERWISE case” will be shown in problem view. And the number of case will be shown in info view.

```
SELECT(RETC1);
  WHEN(01)
      display( 'tower size must be at least 2' );
  WHEN(02) PUT SKIP EDIT
         ('  ERR',IPIC,' TABELLE ATTRIBUTS ZU KLEIN') (A);
  WHEN(03) PUT SKIP EDIT
         ('  ERR',IPIC,' DCF-INPUT FEHLT PANELDATA') (A);
  WHEN(04) PUT SKIP EDIT
         ('  ERR',IPIC,' TABELLE OUTPUT-FELDER ZU KLEIN') (A);
  WHEN(05) PUT SKIP EDIT
         ('  ERR',IPIC,' DCF-INPUT FEHLT PANELKOMM') (A);
  WHEN(06)  PUT SKIP EDIT
         ('  ERR',IPIC,' DCF-INPUT FEHLT THEE') (A);

  WHEN(07) PUT SKIP EDIT
         ('  ERR',IPIC,' DCF-INPUT FEHLT PANELMFS') (A);
  WHEN(08) PUT SKIP EDIT
         ('  ERR',IPIC,' DCF-INPUT FEHLT FORMATNAME') (A);
  WHEN(09) PUT SKIP EDIT
         ('  ERR',IPIC,' DOPPELTE FELDNAMEN IN TA-TAB ',FTEXT) (A);
  WHEN(10) PUT SKIP EDIT
         ('  ERR',IPIC,' STRUKTURSCHACHTELLUNG ZU TIEF') (A);
  WHEN(11) PUT SKIP EDIT
         ('  ERR',IPIC,' INSERTL›NGE NICH NUMMERISCH') (A);
  WHEN(12) PUT SKIP EDIT
         ('  ERR',IPIC,' FELDNAME <',TRIM(FTEXT),
                   '> FEHLT IN DER OUT-TABELLE') (A);
  WHEN(13) PUT SKIP EDIT
         ('  ERR',IPIC,' FELDNAME <',TRIM(FTEXT),
                   '> EXT-ATTR GESETZT OHNE PLATZVORGABEN') (A);
END;
```

Figure 6.24 Sample Code E

Test Results

```
SELECT(RETC1);
    WHEN(01)
        display( 'tower size must be at least 2' );
    WHEN(02) PUT SKIP EDIT
           ('  ERR',IPIC,' TABELLE ATTRIBUTS ZU KLEIN') (A);
    WHEN(03) PUT SKIP EDIT
           ('  ERR',IPIC,' DCF-INPUT FEHLT PANELDATA') (A);
    WHEN(04) PUT SKIP EDIT
           ('  ERR',IPIC,' TABELLE OUTPUT-FELDER ZU KLEIN') (A);
    WHEN(05) PUT SKIP EDIT
           ('  ERR',IPIC,' DCF-INPUT FEHLT PANELKOMM') (A);
    WHEN(06)  PUT SKIP EDIT
           ('  ERR',IPIC,' DCF-INPUT FEHLT THEE') (A);

    WHEN(07) PUT SKIP EDIT
           ('  ERR',IPIC,' DCF-INPUT FEHLT PANELMFS') (A);
    WHEN(08) PUT SKIP EDIT
```

Tasks | PL/I CheckStyle Problem View | PL/I CheckStyle Info View | Clone Tree View | PROCEDURE Call Tree View

!/?	Description	Value	Entity	Threshold	Resource	Path	Location
⚠	GOTO	NOT GOOD	-	-	HelloWorld.pli	/tower/pli/...	line 7
⚠	GOTO	NOT GOOD	-	-	HelloWorld.pli	/tower/pli/...	line 16
⚠	Missing OTHERWISE case	NOT GOOD!	SELECT	-	HelloWorld.pli	/tower/pli/...	line 100
⚠	Over-nested Loop	4	-	3	HelloWorld.pli	/tower/pli/...	line 11
⚠	Too Deep IF statement Depth	Level 3	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 44
⚠	Too Deep IF statement Depth	Level 3	IF_UNIT	2	HelloWorld.pli	/tower/pli/...	line 47
⚠	Too few comments	2.027027%	-	40%	HelloWorld.pli	/tower/pli/...	line 1
⚠	Too Many Blocks and Groups	15	-	10	HelloWorld.pli	/tower/pli/...	line 1
⚠	Too Many Condition Statments	45.0%	-	30%	HelloWorld.pli	/tower/pli/...	line 1

Figure 6.25 Test Result of Sample Code D in Problem View

Tasks | PL/I CheckStyle Problem View | PL/I CheckStyle Info View | Clone Tree View | PROCEDURE Call Tree View

!/?	Description	Value	Entity	Threshold	Resource	Path	Location
ℹ	Loop Depth	4	-	3	HelloWorld.pli	/tower/pli/...	line 11
ℹ	Loop Depth	1	-	3	HelloWorld.pli	/tower/pli/...	line 22
ℹ	Loop Depth	1	-	3	HelloWorld.pli	/tower/pli/...	line 29
ℹ	Loop Depth	1	-	3	HelloWorld.pli	/tower/pli/...	line 32
ℹ	Margin has no problem	GOOD	-	-	HelloWorld.pli	/tower/pli/...	line 1
ℹ	No Foul Words	GOOD	-	-	HelloWorld.pli	/tower/pli/...	line 1
ℹ	Number of bit operation	1	-	5	HelloWorld.pli	/tower/pli/...	line 38
ℹ	Number of Block	15	-	10	HelloWorld.pli	/tower/pli/...	line 1
ℹ	Number of cases in SELECT	13	SELECT	-	HelloWorld.pli	/tower/pli/...	line 100
ℹ	Number of cases in SELECT	5	SELECT	-	HelloWorld.pli	/tower/pli/...	line 132
ℹ	Number of code statements in comment	0	-	5	HelloWorld.pli	/tower/pli/...	line 1
ℹ	Number of comparison operation	1	-	5	HelloWorld.pli	/tower/pli/...	line 25
ℹ	Number of comparison operation	1	-	5	HelloWorld.pli	/tower/pli/...	line 41
ℹ	Number of comparison operation	1	-	5	HelloWorld.pli	/tower/pli/...	line 44
ℹ	Number of comparison operation	1	-	5	HelloWorld.pli	/tower/pli/...	line 47

Figure 6.26 Test Result of Sample Code D in Info View

The analyzer shows the correct answer. The SELECT group is missing OTHERWISE case and it has 13 cases in total. And the messages are displayed in the correct views.

Chapter 7

Evaluation

In this chapter, we are going to discuss how the application is applied in the real situation. In order to evaluate the application if it provides the solutions for BMW group, 4 PL/I programmers and software project managers in BMW are selected to be the evaluators. The results of the evaluation can be the future improvement and references.

Also, this chapter is going to discuss the differences of this application from the others and new features that the others do not have. Although most of the qualities checking tools are specific for Java and C/C++, based on their features and functionalities, we can figure out what make the application extraordinary.

7.1 Users Evaluation

The evaluation process was taken place in BMW group. 4 software developers with PL/I programming experiences are selected to be the evaluators. They are regarded as the potential users of this application. The purpose of the evaluation is to get the general comments from the potential users and to know if the stability of performance can be approved during the tests. The original copies of the survey results are attached in the appendix.

7.1.1 Test Cases and Results

There are 4 cases that request the users to test. And these cases are mainly to start the analysis and modify the rules of analyzers in the preference page, which mainly tests the stability of performance of the application.

Test case 1. To start "PL/I Checkstyle" by selecting a PL/I file and the results should be displayed in PL/I Checkstyle Problem and Info View.

Test case 2. To open the "Checkstyle Preferences" (Window -> Preferences… -> Checkstyle Preferences) and modify the threshold value in LOCAnalyzer to 10, so that checking a file with more than 10 LOC will get a warning message – "Too many LOC". Disable CommentRatioAnalyzer, so that checking a file without a single comment will NOT get a warning message – "Too few comments". And then, do the test case 1.

Test case 3. To start "Procedure Call Tree" by selecting a PL/I file and the results should be displayed in Procedure Call Tree View.

Test case 4. To start “Clone Detection” by selecting a PL/I file and the results should be displayed in Clone Tree View.

All the evaluators can successfully finish the test cases without any errors. And after the results are shown in the views, the introduction of the analysis was presented by the author. The analysis results are interpreted and explained. The evaluators can have an overall idea of how the application works and how the use this application.

7.1.2 Overall Comments

The following comments are the general comments on this application, which can be divided into 4 sections.

Usefulness of the rules and results

Regarding to the usefulness of the rules and results, most of the evaluators thought that the rules are very useful and necessary to be checked in a PL/I program. And the tool is helpful for them. Especially for the rules like OTHERWISE check in SELECT group, checking GOTO and nested loop, they are helpful to assess the quality of a program.

Practicality of the application

Most of the evaluators thought that the application is practical and applicable. It is because the application is integrated with Rational Developer System z (RDz). This is a good approach to the solution. Also, Procedure Calling Analysis provides a practical solution that the PL/I compiler cannot provide this information.

Performance of the application

The performance of application is accepted by most of the evaluators. Especially the Clone Detection, its performance is excellent even if a large amount of codes is analyzed. However, one of the evaluators is skeptical with the performance of the PL/I Checkstyle when an extremely case happens, e.g. 10000 LOC per file. And although this extreme case happens rare, there is some cases happened in BMW.

Improvement and Suggestions

One evaluator suggested that identifier naming should be checked, because there is some meaningless identifier names are defined in the situation of BMW. Moreover, the declaration of built-in function leads a problem of identifier name checking in the application. It has to avoid checking this kind of declaration. Also, for the clone detection, it happens with detecting some SQL static statement which has duplicated

patterns. Therefore, it is suggested that the clone detection in SQL statement should be ignored.

Regarding to the extensible features, one evaluator suggests that it could be an architecture that can allow users to extend the analyzers by using XML file to construct their own analyzers instead of programming.

7.2 Extraordinary Features and Functionalities

The extraordinary features and functionalities of this application are as follows:

- PL/I code quality checking
- Multi-language extensible architecture
- Sophisticated Clone Detection algorithms

PL/I code quality checking

There are many quality-checking tools available either open source products or commercial products; however, they are mainly for Java and C/C++. There is no tool for PL/I, which is difficult language to analyze, as we have discussed it in chapter 2. Therefore, this application is probably the first IDE-integrated Checkstyle tool for PL/I.

Multi-language extensible architecture

Although there are some quality-checking tools that can allow users to extend their applications by adding more checking rules or analyzers, they are not allowed to be extended in other programming languages. In this application, as mentioned in chapter 5, this application has multi-language programming support architecture. Therefore, it can be a COBOL, Java and PL/I Checkstyle application, which can check different codes in the same development environment.

Sophisticated Clone Detection algorithm

The Clone Detection algorithm is mainly from ConQAT. By comparing the duplicated code checking in other quality checking tools, like Checkstyle, Findbugs and PMD, their way to detect duplicated code is not that sophisticated enough. For example, the computational time is not that fast and they do not support fuzzy clone detection (i.e. strict clone detection only). In this application, the clone detection can serve fuzzy clone detection and the computational time is extremely fast, since the time complexity is linear (i.e. O(n)).

Chapter 8

Conclusion

This chapter summarizes the contributions this thesis makes to PL/I Quality Checking tool and provides directions for future work.

8.1 Summary and Contributions

Maintenance activities always consume the largest part of the total cost in a software development process. Therefore, minimizing the cost in maintenance is a critical problem that is mostly concerned by software engineers and software developers. During the software development process, to avoid bad programming practice or style can help to minimize the cost of maintenance. In order to locate the bad programming practice or style, using legacy checkstyle application is one of the ways to achieve the purposes.

Based on the needs of BMW's PL/I software systems, the thesis proposed a list of rules that can restrict bad programming styles or practices in a program. These rules can improve the software maintainability and have economical impact on software development process. Furthermore, based on this list of rules, a prototype quality-checking tool – temporarily called "PL/I Checkstyle" is implemented. Since Rational Developer System z (RDz), which is Eclipse platform base is used in BMW, a tool that can integrate with the development environment would be a good approach to the solution. Therefore, the tool is an Eclipse Plug-in application. This tool has more than 30 rules to be implemented. And this tool has some features and functionalities, which are more advanced than the similar open source tools available in Internet. The extraordinary features are as follows:

- PL/I code quality analysis
- Support multi-languages and analyzers extension
- Sophisticated clone detection algorithm

The prototype is also tested and evaluated by potential users, who are mainly the software engineers and PL/I experts from BMW. In general, the feedbacks are positive. The evaluators thought that this application is useful and helpful during the development process. They also suggest some improvements and ideas that can add values in this tool. Their ideas can be references of the future works.

All in all, the tool can help to improve PL/I code quality, reduce the cost of maintenance and increase the productivity in software development process.

8.2 Future Works

In the future, the enhancement of the tool can be divided into 3 aspects.

The first aspect is optimization of the performance. In order to minimize the time complexity of the analyzers, better algorithms have to use in the analyzers. Also, the accuracy of the analysis results has to be improved.

Secondly, based on the suggestions by the evaluators and the needs of BMW software development process, more analysis rules or analyzers should be added, for example, identifier-naming checking, which checks if the name of variables or identifiers is meaningful or not. Also, a full integration of the tool in the BMW mainframe system is needed.

Finally, different programming languages code analyzers can be added in the tool. A multi-language quality-checking tool can be made. Also, more extension points could be created to maximize extensibility that allows extending the tool.

References

1. http://checkstyle.sourceforge.net/
2. http://findbugs.sourceforge.net/
3. http://pmd.sourceforge.net/
4. http://www-306.ibm.com/software/awdtools/rdz/
5. B.W.Boehm, Software Engineering Economics. Prentice Hall, Englewood Cliffs, N.J. , 1981.
6. International Standard Organization. ISO 9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use, Dec. 1991.
7. F.Deißenböck, M.Pizka, T.Seifert. Tool-supported realtime quality assessment. In Ying Zou and Massimiliano Di Penta, editors, *Proceedings of the 13th International Workshop on Software Technology and Engineering Practice 2005 (STEP-05)*, pages 127-136. IEEE Computer Society, September 2005.
8. Markus Pizka and Florian Deissenboeck. How to effectively define and measure maintainability. In Ton Dekkers, editor, *SMEF 2007 - 4th Software Measurement European Forum*, number ISBN 9-788870-909425, Rome, Italy, May 2007.
9. B. W. Boehm et al. Characteristics of Software Quality. North-Holland, 1978.
10. IEEE 1219 Software maintenance. Standard, IEEE, 1998
11. F. Deissenboeck and M. Pizka. The economic impact of software process variations. In *International Conference on Software Processes 2007 - Software Process Dynamics and Agility*, number 4470 in Lecture Notes in Computer Science, Minneapolis, MN, May 2007. Springer.
12. http://www.misra.org.uk/
13. Vic Hartog and Dennis Doomen. Coding Standard: C# *Philips Medical Systems - Software / SPI* . Philips' proprietary, © 2003 Philips Electronics N.V.
14. http://en.wikipedia.org/wiki/PL/I#Search_for_a_string
15. http://home.nycap.rr.com/pflass/plistyle.htm
16. D.Harms, PL/I Programmierrichtlinien für LTERM, 2007
17. Office 2004 Test Drive User, M.Pizka, , itestra GmbH
18. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html
19. http://www.ibm.com/developerworks/java/library/os-ecplug/
20. http://www.eclipseplugincentral.com/
21. Pressman, Scott. *Software Engineering: A Practitioner's Approach*. Sixth Edition, International, p 388. McGraw-Hill Education 2005.
22. Crosby, P., *Quality is Free*, McGraw-Hill, 1979

23. **McConnell, Steve. *Code Complete* First Ed, p. 558. Microsoft Press 1993**
24. **BZ Research, Java Use and Awareness Study, Dec.2005**
25. **Edsger Dijkstra (March 1968). "Go To Statement Considered Harmful". Communications of the ACM 11 (3): 147–148.**
26. **http://conqat.cs.tum.edu/index.php/CloneDetectionTutorial#Clone_Detection_in_ConQAT**
27. **http://conqat.cs.tum.edu/index.php/Main_Page**

Appendix A

LPEX

LPEX (Live Parsing Extensible Editor) is a programmable editor that can be used to create and edit many kinds of files, including programs, documentation, and data files. In addition to basic editing functions, the editor also offers the following features:

- Language parsing uses automatic indenting, color, and text effects to emphasize different parts of your source program, such as programming language keywords, logic structures, comment lines, and arithmetic operators.
- Locations marking facilities let you define *bookmarks* that let you move quickly from one location to another within your source file.
- Elaborate search facilities let you specify the precise scope of the search. In addition to locating specific character strings, editor search facilities will also accept regular expression search strings that can finding matching search *patterns* in your source file.
- Multiple editing views let you have more than one file open for editing at a time.
- Multiple editing windows let you open multiple views of the same file for editing, letting you view different parts of the file at the same time. Changes to the file in any view are reflected immediately in other views containing that same file.
- An Editor programming language lets you program elaborate macros to automate recurring editing tasks.
- Keystroke recording facilities let you *record* sets of keystrokes, which you can later replay or incorporate into your own macros.

The following components are mainly used in the prototype.

LpexParser

Interface LpexParser can be implemented to define a document parser. A parser is associated with one particular view of a document. The parser must have a constructor whose parameter is LpexView. In the Lpex editor core package, the implementing class is **LpexCommonParser,** which provides the basic functions to analyze a document.

LpexCommonParser

LpexCommonParser is a base, abstract class for document parsers. It adds several services to the LpexParser interface, simplifying document-parser development, and provides methods for a common look-and-feel to the parsers extending it. Certain methods in the basic LpexParser interface are not used by parsers extending LpexCommonParser.

Life cycle

This is the life cycle of an LPEX document parser:

- The parser class is loaded in the editor
- The parser constructor is invoked
- The initialize method is called in an LpexCommonParser-based parser
- Parser's total parse method is called for an initial parse of the document view
- The parser is called to parse incrementally every time a change to the document is completed
- The parser is notified of its upcoming termination
- Its class is unloaded.

The process above takes place for each edit session of a document view, whenever there is a parser associated with that document type or a parser is explicitly specified. It also takes place any time the **updateProfile** command is run in a view, either called explicitly or as a result of other actions (such as a document reload).

This makes it important to handle the parser termination and reinstantiation correctly, especially when there are other resources and tools initiated from, or associated with, the document parser. Also, document models that are updated solely on the basis of information provided by the parser calls, should be refreshed when a document parser is (re)instantiated.

Total and incremental parse

There are two main callback methods in a document parser:

- `parseAll()` - total parse: parse the entire document. This method is normally called when a new document has been loaded in the editor, whenever the **updateProfile** command is run, and when a total parse is activated directly (with, for example, the **parse all** editor command).

- `parseElement(int element)` - incremental parse: parse changes to the document. This method is normally called when a change has been committed in the editor (for example, the text on a document line was edited and then the cursor was moved off that line), and when an incremental parse is activated directly (with, for example, the **parse** editor command). The **element** parameter indicates an element whose committed change triggered the parse, or the element that precedes or follows a deleted block. The parser may have to identify other neighboring elements that will have to be reparsed as one unit, for example an element that is part of a multi-line javadoc comment in a Java source document. In addition, for performance reasons, a parser should check whether neighboring elements have changed (in the case that the incremental parse was triggered, for example, by a multi-line paste or block operation), and parse them as one unit during the same call, eliminating the overhead of additional parseElement() callbacks. This is accomplished with the `parsePending()` and `elementParsed()` methods.

Changes to the document carried out during parsing are not recorded in the parse-pending list of elements.

Parser properties

Parser properties are parser-specific settings. Certain common parser properties are used by LpexCommonParser and are, therefore, in effect in all the parsers that extend it:

- **autoIndent** - to enable/disable auto-indenting (if not specified, assumed to be "on")
- **errorMessages** - to turn on/off the embedding of parser error messages (via the addMessage() methods) (if not specified, assumed to be "on")
- **maintainBidiMarks** - convenience, document-scoped property to indicate whether bidi marks should be maintained by the primary document parser during incremental parsing in documents with ignorable bidi marks; use the **document.** prefix to set and query this property (if not specified, assumed to be "off").
- **parseAfterEveryKey** - to optionally override the **parseAfterEveryKey** parameter setting in effect in the editor.
- **proto.*keyword*** - keyword expansion template for the **proto** action
- **style.*styleCharacter*** and **styles.backgroundColor** - used, for instance, by preference pages that customize parser styles; see `setStyle()`
- **styleName.*styleCharacter*** - used, for instance, by preference pages that customize parser styles; see `getStyleName()`
- **tokenHighlight** - to turn token highlighting (syntax coloring) on/off (if not specified, assumed to be "on"); see also `setStyle()`
- **wordBreaks** - to define word-break criteria for the word navigation actions **nextWord**, **prevWord**, **blockMarkNextWord**, and **blockMarkPrevWord**; one or more of the following parameters can be used:
  - **caseChange** - break on lower to upper case changes
  - **defineChars** - define the characters that are considered part of a word. The default definition of a word in LPEX for the purpose of the word navigation editor actions is any segment of characters delimited by whitespace; this parameter changes this definition to what the method `isWordCharacter()` defines, by default all letters and digits (this method may be further extended or overridden by the parser)
  - **fieldStart** - break at the start of an editing field
  - **nonWordChars** - break on non-word characters (also stops on start and end of lines)
  - **styleChange** - break on a change in style (new token).

The default scope of a parser property is the document parser for which it is defined. Any instance of the parser in any document view will use the same property value, unless overridden by a document- or view-scoped setting. Document-scoped parser properties are prefixed with `"document."`. View-scoped parser properties are prefixed with `"view."`. For example, the value of the **autoIndent** setting used by a specific instance of a document parser may be changed by the **view.autoIndent** parser property.

Parser properties may be defined at several levels.

- The installation settings of parser properties are defined in the parser profile. An example is the file Profile.properties, a C/C++ document parser profile.
- Default settings, which override the installation settings, are stored in the editor defaults profile (see the **defaultProfile** editor parameter).
- View settings, which override the default settings for a particular parser instance, are stored in the associated document view.

Most parser properties are also accessible through the **parserProperty** editor parameter. A parser may listen to changes effected to any of its properties by extending `propertySet()`.

See `getProperty()` and `setProperty()` for additional information on parser properties.

Language-sensitive help

LpexCommonParser provides a few basic services for the implementation of language-sensitive help (LSH) on any platform:

- `getLshToken()` determines the current token in the edit area. This is done based on the text at the current document position (cursor), or any specific text selection in effect in the view. The token returned may then be used as the key into a table that maps language keywords to particular help panels.
- Document parsers may extend `getLshToken()` to further refine the generation of the key which will be used to access the help-mapping table. This is needed in those cases in which the cursor is located in particular contexts inside the document, such as comment lines, identifiers, embedded error messages (displayed as **show** lines), and so on, where the token returned by method `getLshToken()` is not a recognized (reserved) keyword in the language. The COBOL parsers, for example, extend `getLshToken()`.
- Document parsers which implement other LSH schemes (such as particular help directly addressing the specific document source code, for example help based on information available in a code store which was created by parsing an entire project) may use `getToken()` instead, to simply retrieve the current token or selection.

LpexView

Use this class to manage an LPEX document view. It essentially provides all the editor widget programming support.

Creating an instance of this class creates a document view. When you're just extending an existing LPEX-based application, view management (creation, disposal) is usually that application's resposibility.

Lines and elements

The LPEX document maintains a list of elements. An **element** in this list constitutes any line of textual information that may be displayed in the edit area of a view of the document. Elements are:

- text elements (lines) - each text element represents one line (or record) in the actual document (and its underlying file). The current line number is displayed on the LPEX status line
- show elements - show elements are typically used by document parsers and their associated tools to present informational and error messages. Each show element in the document list of elements is created for, and belongs exclusively to, one particular view. Show elements are not saved with the document.

Most methods that access the document contents use as argument an element ordinal number. The ordinal numbers of elements change as elements are being inserted and removed, as a result of, for example, the user editing the document and parsers adding and removing error messages.

In general, a document parser skips and ignores show elements:

```
if (view.show(elementNumber))
 {
  // embedded message - skip it
 }
else
 {
  // text line - parse it
  // . . .
 }
```

LpexCharStream also ignores show elements, and does not send them to the token manager.

During the processing of one parse unit (i.e., during one total-parse or one incremental-parse call from LpexCommonParser), requests to `addMessage()`,

`removeMessages()`, `addDocumentMessage()`, and `removeDocumentMessages()` are stacked, and only processed afterwards. In other words, during one such call, the parser will see the same element numbers throughout the document - the messages will only be added and removed by LpexCommonParser when the parser returns.

In view of the above, a parser should not concern itself with changes in element numbers during a parse operation. If, however, your parser keeps (caches) information across parse calls from LpexCommonParser, you may consider keeping it in term of lines rather than elements. To determine the document line number of a text element, you can use `lineOfElement()`. Marks may also be set to tag particular elements for this purpose (see the **mark** editor parameter).

If your code itself inserts a series of show lines (for example, compiler error messages) in a view, outside a parsing operation, the easiest way to do it is starting from the last (i.e., from the highest element number), and proceeding towards the first (lowest element number).

Appendix B

Since there are several light-weight parsers in the prototype PL/I Checkstyle tool, the following are the components that construct the parsers. Of course, before using the following analyzers, `PliTokensGenerator.java` has to be used to get a list of tokens from a program. The pseudo-code is used to explain the implementation details.

Block Analyzer

This is a class that extracts the block or group in a source file. The block includes PACKAGE, PROCEDURE and BEGIN; the group includes DO and SELECT. Therefore, for getting the block and group in a program, it has to be done by following algorithm.

```
Get a list of tokens in the program
Create a BLOCKPAIR list for storing token pairs

FOR each token in this list

     IF the token is PACKAGE THEN
           find the END token for PACKAGE
           put this pair of tokens into the BLOCKPAIR list
     END IF

     IF the token is PROCEDURE THEN
            CALL matchToken method to find the END token for PROCEDURE
            put this pair of tokens into the BLOCKPAIR list
     END IF

     IF the token is BEGIN THEN
            CALL matchToken method to find the END token for BEGIN
            put this pair of tokens into the BLOCKPAIR list
     END IF

     IF the token is DO THEN
             CALL matchToken method to find the END token for DO
             put this pair of tokens into the BLOCKPAIR list
     END IF

     IF the token is SELECT THEN
             CALL matchToken method to find the END token for SELECT
             put this pair of tokens into the BLOCKPAIR list
      END IF
END FOR
```

Block Depth Analyzer

After getting the blocks and groups from BlockAnalyzer, the depth level of a block can be calculated by comparing their location. The following algorithm can get the depth level of blocks and groups.

```
Get the BLOCKPAIR list from BlockAnalyzer

FOR each of the block pairs in the BLOCKPAIR list

     GET block start location and STORE it in BLOCKSTART_1
     GET block end location and STORE it in BLOCKEND_1
     SET COUNT = 1

     FOR each of the block pairs in the BLOCKPAIR list

           GET block start location and STORE it in BLOCKSTART_2
           GET block end location and STORE it in BLOCKEND_2

           IF BLOCKSTART_1 and BLOCK_END_1 is
           within BLOCKSTART_2 and BLOCKEND_2 THEN

                COUNT = COUNT +1

           END IF
       END FOR

       PUT COUNT into BLOCKPAIR as the depth of corresponding block

END FOR
```

IF-UNIT Analyzer

Since IF-statement is not regarded as a block or a group in PL/I, it makes IF-statement or nested IF-statements difficult to be analyzed. Therefore, IF-statement should be distinguished as several units: IF-UNIT, ELSE-UNIT and ELSE-IF UNIT. The following pseudo-code is the implement of this analyzer.

```
GET a list of token in a program file
Create a CONDITION_BLOCK list to store IF units.
FOR each of the tokens

          IF it is “IF” but the previous token is not “ELSE” THEN
              CALL findEndToken to find the END token
              Classify as “IF-UNIT” and store this IF unit into
              CONDITION_BLOCK
          END IF

          IF it is “ELSE” but the previous token is not “IF” THEN
              CALL findEndToken to find the END token
              Classify as “ELSE-UNIT” and store this ELSE unit into
              CONDITION_BLOCK
          END IF

          IF it is “ELSE” and the next token is “IF” THEN
                CALL findEndToken to find the END token
                Classify as “ELSEIF-UNIT” and store this ELSEIF unit into
                CONDITION_BLOCK
          END IF

END FOR
```

IF-UNIT Depth Analyzer

This analyzer can get the depth level of IF-unit which is given by IF-UNIT Analyzer. The algorithm of this analyzer is implemented same as the Block Depth Analyzer. Therefore, it would not be discuss in here.