Chair of Software Engineering

Rule-Based Code Analysis

Master Thesis

Stefan Zurfluh
ETH Zurich
zurfluhs@student.ethz.ch

October 1, 2013 - April 1, 2014

Supervised by:
Julian Tschannen
Prof. Dr. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Program analysis – static or dynamic – is an important method to improve and maintain code quality. Static analysis gives immediate feedback during the development of a computer program, and can be a great help for programmers. The language design of the Eiffel programming language already encourages high code quality, but the Eiffel IDE has no light-weight static analysis tool.

We designed a versatile and extensible rule-based framework for light-weight static program analysis for the Eiffel language called Inspector Eiffel. We implemented a basic set of rules and developed a user interface that is integrated in EiffelStudio, the main Eiffel IDE. The tool is fully usable by programmers and integrates well in the development process. We present several case studies of applying the tool on existing libraries and programs, which show the usefulness of the tool by detecting coding issues and suggesting improvements.

Acknowledgments

I would like to thank my supervisor Julian Tschannen very much for his continuous outstanding help during my whole thesis project. Many thanks go to Prof. Dr. Bertrand Meyer for his great support. As well I want to thank Đurica Nikolić, Mischael Schill and many others from the Chair of Software Engineering for their helpful comments.

Then I thank the people from *EiffelSoftware*, especially Emmanuel Stapf, Alexander Kogtenkov, and Jocelyn Fiat, for their many suggestions and comments they made regarding the analyses, and for their help in regard to *EiffelStudio*.

Stefan Zurfluh

Contents

Chapter 1

Introduction

Achieving and maintaining high code quality is one of the biggest challenges in software engineering [6, 11]. Very often it does not suffice to have some important practices in mind. Many times, having a written document with coding guidelines [5] does not work in practice either. Checking code against a document is tedious and time-consuming. Here, automated tools aiming at high code quality come into play [2, 3].

Because they need only little user input and they point out the flaws in the code to the developer, automated tools are both efficient and convenient from the perspective of the software developer. The effort needed by the developer is reduced, which encourages the explicit and continuous improvement of code quality. Some tools are fully automatic, meaning that running them both diagnoses problems and fixes them immediately. The tool we developed is partially automated: The diagnosis part runs automatically, while the decision to change the code is left to the user. Changes can be made manually or automatically. Human judgment is inevitable in connection with our method.

Our approach uses a framework and a tool for *static code analysis* called Inspector Eiffel. It is useful for improving code quality in many different contexts and on many different levels. It can be used to detect potentially dangerous runtime behavior. It may also be used to enforce a consistent coding style. It is capable of suggesting code patterns and alternative ways of coding to the programmer.

In Chapter 2 we present Inspector Eiffel for EiffelStudio in detail. Inspector Eiffel is the result of this thesis project and is its main outcome. We explain the method we pursued. Then the rules are discussed. They form an essential part of the concept of Inspector Eiffel. We also go through the various parts of the user interface, covering both command-line and GUI usage.

Results from applying Inspector Eiffel to a large amount of code is shown in Chapter 3. We have analyzed two fundamental libraries used in Eiffel systems: *EiffelBase* and *EiffelVision*. From the code of EiffelStudio (which has a seven-digit number of lines of code) we analyzed only selected parts. All three projects mentioned have been released under an open source license. Moreover, we made

an analysis of our own code, the results of which are shown in a further Section.

Chapter 4 deals with the implementation of the INSPECTOR EIFFEL framework and tool. We present key parts of our software design as well as important interfaces. In addition to that, it is shown how rules are checked by the code analyzer. Then, we present two examples where we show in detail how a rule is implemented. We also dedicate a section to adding new rules, including more complicated ones. The implementation of the user interfaces (graphical and command-line) is discussed in a further section.

We conclude our thesis by mentioning possible future work and related work.

The appendix consists of a comprehensive list of the rules that were proposed during the project.

Chapter 2

Inspector Eiffel

2.1 Method

2.1.1 Framework

The Inspector Eiffel framework for Eiffel is designed in a manner that allows for extensibility and customizability. Any analysis that the framework contains is based on a *rule*. The framework currently contains a set of more than 30 rules (see Appendix A). The rules range from simple pattern-matching to sophisticated control-flow analysis. The Inspector Eiffel framework is implemented as an Eiffel *library*, a reusable component which can be included in any Eiffel system.

Every compiler performs some static program analyses, an example being static type checking [1, 13]. The Eiffel compiler already includes analyses that are not syntactically required: warnings are displayed after compilation, such as when a local variable is not used. Our framework is separated from the compiler but not fully independent. We retrieve the abstract syntax trees and other information about the analyzed classes from the compiler. Therefore a necessary requirement for making a code analysis is that the code was compiled successfully. Then, during analysis, the interfaces to the Eiffel compiler are used by Inspector Eiffel. Like this we can avoid redundancies: we can avoid to do computations that have already been done by the compiler.

Operation

From an external viewpoint, the operation of Inspector Eiffel is described as follows: Inspector Eiffel takes as an *input* a set of *classes* and produces as an *output* a set of *rule violations* for these classes. The input classes must be a part of the Eiffel system or of a referenced library. Yet it is possible that some of them are not compiled since they are not referenced (transitively) in the code of the system. These classes are skipped and are not analyzed.

The API of Inspector Eiffel directly corresponds to the class input and

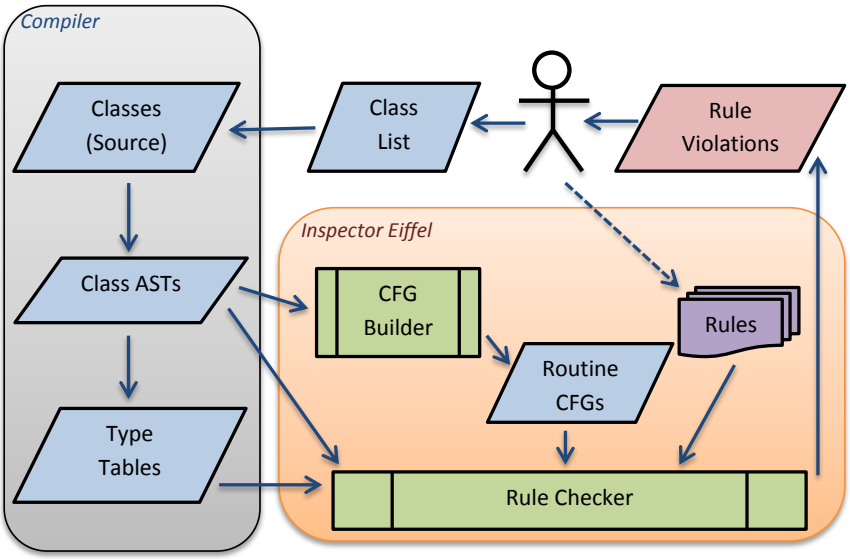

Figure 2.1: Data flow in Inspector Eiffel.

the rule violations output. Some other functionality was added for convenience. (See Section 4.1 on API implementation.)

Internally, there is more data flow than what is visible from the user side. Figure 2.1 shows a schematic representation of the data flow during code analysis. The Eiffel compiler plays an essential part in preparing the code analysis. The compiler provides Inspector Eiffel with the abstract syntax trees (ASTs) of all the classes from the user's list that are compiled. In addition, Inspector Eiffel requests type information from the compiler for all the classes. This is needed since the AST of the Eiffel compiler does not contain complete type information. Apart from the AST and the type table Inspector Eiffel is in need of the control flow graph ("CFG") for every feature of each class. The control flow graph is computed by a module of Inspector Eiffel.

A list of rules is available to Inspector Eiffel. The user is able to disable or enable rules and to set options for individual rules and for Inspector Eiffel in general. What is most important in the context of data flow is that all the *enabled* rules are used to check the classes for rule violations. We defined two types of rules: *standard rules* and *control flow graph rules*. *Standard rules* operate on the AST only, and they have the type information available. *Control flow graph* rules, operating on the control flow graph, can also access the type information and, if needed, they can access the abstract syntax tree.

Each active rule creates a list of rule violations. Inspector Eiffel collects the violations of all rules and outputs them to the user.

Checking Rules

We will explain briefly our method of analyzing classes with regard to rule violations. We denote this process as *rule checking*. The method for a single class will be explained; multiple classes are analyzed sequentially, so in this case the procedure would be repeated.

Standard Rules When initialized, every standard rule notifies INSPECTOR EIFFEL of the AST node types the rule needs to process. There is a *rule checker* module that keeps track of these AST processing actions. Now, for analysis, the module iterates over the AST of the class *only once*, which increases performance as compared to an iteration for each rule. At each node all the rules that registered an action for the corresponding node type, are notified. At the end of the AST iteration all standard rules have completed their analysis.

Control flow graph rules The control flow graph is created for each feature defined in the class. Usually a control flow graph rule does a fixpoint iteration over the graph edges using a worklist algorithm. The rule checking is complete when the algorithm has reached a fixpoint.

Fixes

Some violations of rules can be fixed automatically. This fact is taken into consideration by the possibility to attach one or more fixes to a rule violation. It should still be left to the user whether to fix a rule violation, so fixing should be a further step after outputting the rule violations to the user interface.

2.1.2 Interface

We designed a graphical tool for the *EiffelStudio* integrated development environment. Like many other tools it appears as a *panel*, i. e., a movable and dockable child window. It essentially consists of two components: a toolbar and a table that can be filled with rule violations. The tool panel closely interacts with the editor. One can navigate through the violations and the editor instantly navigates to the corresponding source code location. Many elements in the table are interactively connected to the editor and to other functionality provided by the EiffelStudio API.

There is limited caching of rule violations so that consecutive analyses do not need to consider unchanged code multiple times, and the tool also supports automatic fixing of rule violations.

2.2 Rules

Since INSPECTOR EIFFEL is *rule-based* it is important to clarify what we mean by a *rule*. In the context of INSPECTOR EIFFEL, a rule can be understood as

(a) a certain kind of static code analysis, or

(b) a certain property that must hold for the program code or for specific parts of it.

Being very precise one could argue that (b) defines what a rule is and (a) is merely the means of checking that the rule holds. This makes sense, however in the implementation this distinction is not upheld for practical and design reasons, as we will see in Chapter 4. Thus in the following we will use the term *rule* interchangeably for both the property and the analysis.

If the source code violates a rule (i. e., does not follow what the rule prescribes) then we call it a *rule violation*. A rule violation describes:

1. The rule that is violated;
2. Its exact location in the program code (or in some cases perhaps only the affected class or feature);
3. Optionally, the name(s) of the affected variable(s) and other relevant data;
4. Optionally, one or more possible ways to fix the problem.

2.2.1 Classification

When we were composing the list of possible rules that Inspector Eiffel could contain, we classified the rules by the following criteria: severity, scope, and applicability.

Severity

To each rule we assigned one of the following four *severity categories*: *error*, *warning*, *suggestion*, or *hint*. The severity of a rule describes how serious a rule violation is and suggests how a violation should be treated.[1]

Error An error indicates code that is very dangerous to execute. Compilation should be aborted, or the program should not be executed. Code for which the code analyzer found errors may be very critical.

Warning A warning indicates that there is code that may lead to dangerous program behavior. Unlike errors, the program may be executed despite of warnings. In certain cases human judgment can show that dangerous behavior will not arise or is very unlikely to arise.

Suggestion A suggestion advises that the code should be corrected. However in most of the cases, such code will not lead to dangerous program behavior. For example, one might expect a performance decrease. Bad coding style often falls into this category, too.

[1] The rule severity has been implemented as described here (four categories). This implementation may be extended or adapted, though.

Hint A hint is only an insignificant rule violation. Hints may for example suggest different ways of coding. In some cases, hints may even be bi-directional in the sense that if such a violation is corrected as the hint proposes then one gets a hint that proposes to change the code back to the old state.

Scope

The scope of a rule indicates roughly how much code has to be analyzed at once in order to detect a rule violation. We clustered the rules into the scopes *instruction*, *feature*, *class*, and *system*. Here is an overview of what each of the scopes stands for:

Instruction scope The rule analyzes certain kinds of instructions individually. It does not need to consider interconnections between multiple instructions. Note however that such instructions may be complex and may contain many other instructions. Any rule that checks for simplifiable `if` instructions is such an example.

Feature scope The rule needs to analyze a whole feature in order to determine whether there exists a violation. E. g., a rule that checks for unused feature arguments has *feature scope*.

Class scope The rule analyzes multiple features or refers to other class-wide code properties. A good example is the rule that checks for very big classes (#33).

System scope The rule analyzes code structures that stretch across several classes. This is the case for the *Feature never called* rule (#3).

Applicability

Most rules apply to *all* kinds of classes. Some rules however should be limited to either *library* classes or *non-library* classes.

Library class rules It is imaginable to have rules that want to enforce a particularly strict and good coding style only analyze classes from a library. Such classes may be used in many different projects. Here, there may be a need for coding standards that are even higher than for normal classes. Therefore one may want special rules that deal with library classes only.

Non-library class rules Non-library class rules *exclude* all library classes. For example, one rule checks for features that are never called. For many classes a feature that is not used indicates some mistake, and this feature should be suggested to be removed. Many features from libraries however are not called and this is normal, even in the context of a large system that uses this library. Of course such features must remain in the library.

The current implementation of Inspector Eiffel requires the programmer to manually mark library and non-library classes. Classes that are not marked will be analyzed by rules of any applicability.

2.3 User Interfaces

Inspector Eiffel is integrated in EiffelStudio. There is no stand-alone version of Inspector Eiffel. After all it uses the EiffelStudio compiler and many other parts of the EiffelStudio API. Moreover, an Eiffel system must compile without any error in order to be allowed for analysis.

There are two ways of running Inspector Eiffel:

1. Using command-line arguments with the command-line version of EiffelStudio. The command-line version of EiffelStudio is run by starting the same executable as for the GUI version, with the *-gui* argument omitted. An Eiffel system must be provided as an argument. This system is compiled (if necessary) and then analyzed. The output of the analysis will be directed to the terminal window.

2. Running the tool in the Graphical User Interface of EiffelStudio. In the GUI, Inspector Eiffel appears as a panel (a movable and dockable tool window). The panel is mainly used for displaying analysis results. The command to carry out an analysis can be found not only in the panel but also in various context menus. Like that, analyzing specific classes or sets of classes is very straightforward and easy.

2.4 Command-Line Mode

2.4.1 Execution

In order to run Inspector Eiffel in command-line mode the EiffelStudio executable *ec.exe* (on Windows systems) or *ec* (on Unix systems) must be launched, which can be found in the *bin* subfolder of the EiffelStudio program directory. We only present the most important options that are usually needed in conjunction with Inspector Eiffel.

The following command will perform a code analysis:

`ec.exe -config` *`projectfile`* `[-target` *`target`*`] -code-analysis [-cadefaults] [-caloadprefs` *`preffile`*`] [-caclass` *`CLASS1 CLASS2 ...`*`]`

The arguments in brackets are optional.

`projectfile` The *.ecf* file of the project to compile and analyze.

`target` The specific system target to act on. If `-target` is omitted and there is more than one target defined in the project, then a user prompt to choose the target is shown before compilation.

`-cadefaults` If provided, all preferences regarding INSPECTOR EIFFEL will be reset to their default values (before this analysis is run). For example, this leads to enabling all rules that are enabled by default, even though they may have been disabled in the GUI before.

`-caloadprefs` Use preferences from `preffile`, an XML file containing INSPECTOR EIFFEL preferences. `preffile` can be generated by exporting the current preferences in the GUI.

`-caclass` Followed by a list of class names (without file extension *.e*) of the classes that shall be analyzed. If omitted, the whole system will be analyzed.

EiffelStudio will try to compile the system if needed. INSPECTOR EIFFEL will only work with syntactically correct code and a compiled system. Should the compilation fail, EiffelStudio will abort and INSPECTOR EIFFEL will not start.

2.4.2 Output

Upon a successful compilation either the whole system or the classes mentioned in the arguments will be analyzed. The class that is currently being analyzed will be displayed so that the user can follow the progress.

As soon as everything needed has been analyzed the results will be displayed as a list of rule violations. These rule violations are sorted by class and by location. In addition to the name of the violated rule and the rule ID, a description of the concrete violation will be displayed as well.

2.5 Graphical User Interface

2.5.1 Running Inspector Eiffel

There are several different ways of running INSPECTOR EIFFEL. Most importantly, you can select the scope of the analysis. For example it is possible to analyze single classes, or whole systems. Also for the same scope there are different ways of running INSPECTOR EIFFEL depending on your personal preferences.

Analyzing the System

If you want to analyze the *whole system* of the currently open project press the *Analyze System* button on the toolbar of the INSPECTOR EIFFEL panel. Every *compiled* class of the system will be analyzed.

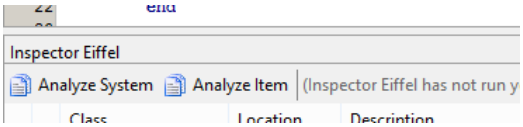

Figure 2.2: The buttons in the tool panel.

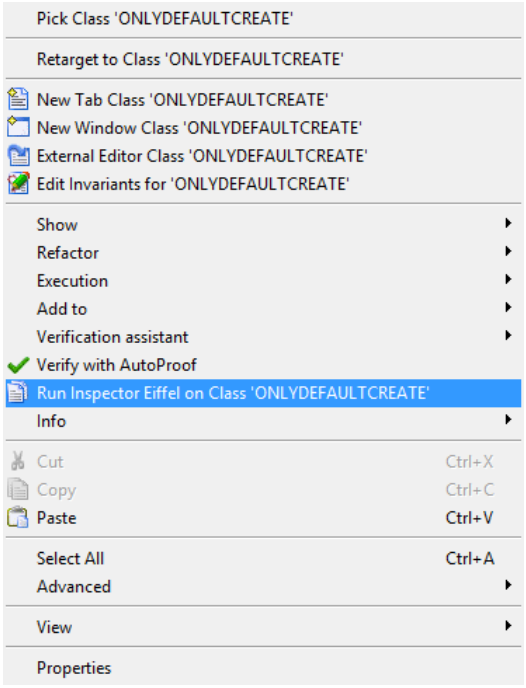

Figure 2.3: The class context menu.

Analyzing a Class or Group

Current Class Left-clicking the *Analyze Item* button on the toolbar of the Inspector Eiffel panel starts an analysis of the class that is *currently open in the editor*.

Any Class There are two ways of analyzing an arbitrary class (either from your system or from a library):

1. Right-click its class name (anywhere you find it) and select *Run Inspector Eiffel on Class "..."* from the context menu (see Figure 2.3).
2. Pick a class and drop it on the *Analyze Item* button in the panel.

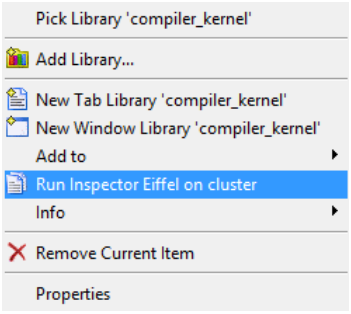

Figure 2.4: The cluster context menu.

Clusters To analyze a cluster there are two possibilities, like for classes:

1. You can either right-click the cluster and select *Run Inspector Eiffel on cluster* from the context menu (see Figure 2.4).
2. Pick a cluster and drop it on the *Analyze Item* button in the panel.

Other Groups Not only clusters but any group (such as a library) can be analyzed by pick-and-dropping it on the *Analyze Item* button in the panel.

2.5.2 Using Analysis Results

Sorting and Filtering

The list of rule violations can be sorted by any column by clicking on its header. Click it again to switch the sorting direction. You can hide and show errors, warnings, suggestions, and hints by clicking the corresponding toggle buttons in the middle of the toolbar. Typing in the text field on the right side of the panel toolbar filters the results. The filter takes into consideration the title, the ID, the affected class, and the description of the rule. It is a live filter that filters while you are typing. Press the button on the right of the text field to clear the filter and display again all violations. Figure 2.5 shows Inspector Eiffel displaying results.

Navigating Through the Results

In the list of rule violations, formatted elements like classes and features are clickable and draggable like anywhere else. In order to navigate to a specific rule violation just double-click the corresponding row. The corresponding class will be opened (if needed) and the cursor will jump to the exact location of the rule violation (some violations do not have an exact location because they refer to a whole class). You can also navigate through the results using the *Go to next rule violation* and *Go to previous rule violation* buttons on the right side of the panel toolbar.

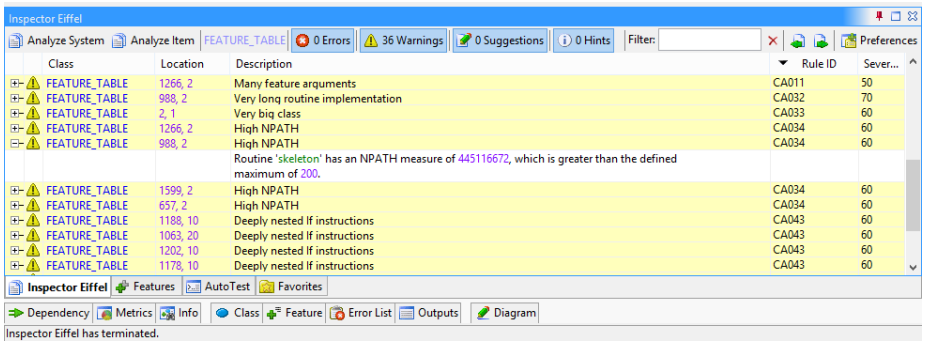

Figure 2.5: The results of code analysis as a list of rule violations (example).

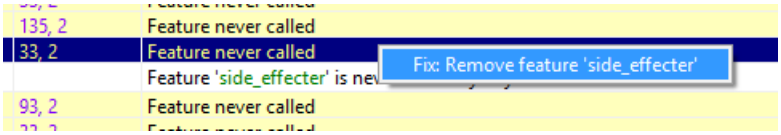

Figure 2.6: Fixing a rule violation.

Fixing Rule Violations

Some violations provide automatic fixing. Right-clicking the corresponding row in the tool panel opens a context menu where you can choose from one or more possible fixes (shown in Figure 2.6). When you click on *Fix: "..."* the source code will be adapted and the project will be recompiled.

Exceptions During Analysis

In case of a bug in a rule, which leads to an exception being thrown during analysis, the exception is caught by Inspector Eiffel. It will show up as an *error* on the very top of the list in the panel, above all rule violations. You can double-click on the entry to see the exception details (the call stack, which rule caused it, and so forth).

When an exception occurs while a class is being analyzed Inspector Eiffel continues with the next class. Despite of exceptions Inspector Eiffel tries to analyze as much as possible. However, some rule violations (of bug-free rules as well) may be missing in this case.

2.5.3 Customization

General Preferences

The *Preferences* button in the panel toolbar opens a dialog containing all preferences for Inspector Eiffel. There you can enable and disable all rules of a

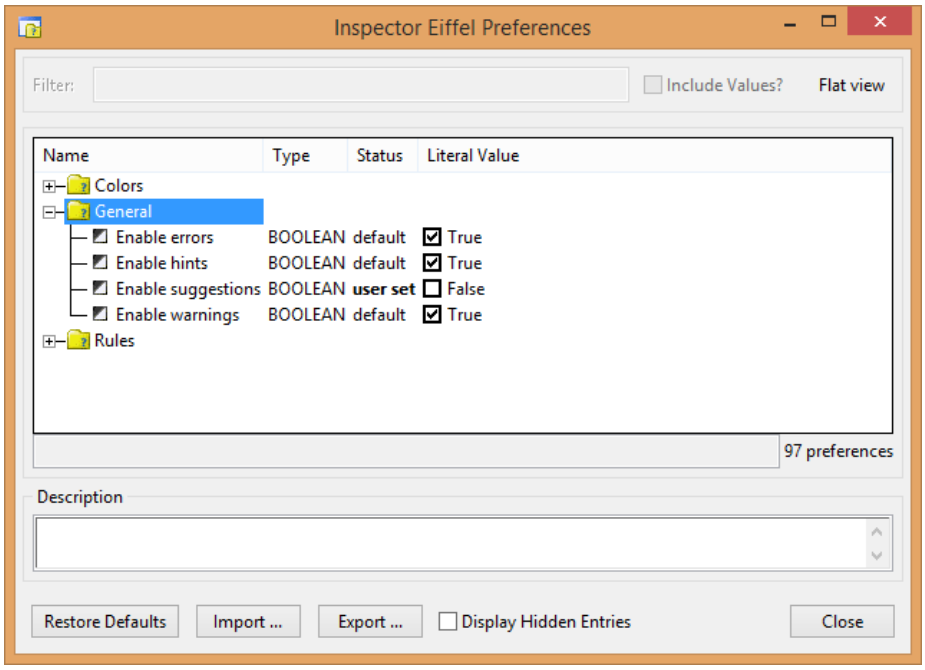

Figure 2.7: The preferences dialog of INSPECTOR EIFFEL.

certain severity, you can choose colors for the results, and there are many preferences that control individual rules. Figure 2.7 shows the preferences dialog for INSPECTOR EIFFEL.

Rule-Specific Preferences

The rule-specific preferences are located in the *Rules* subfolder (shown in Figure 2.8). Two preferences can be found for every rule: *Enabled/disabled* and the *severity score*. Some rules have additional integer or boolean preferences like thresholds.

Exporting and Importing Preference Profiles

Using the buttons in the preferences dialog one can export these preferences to an XML file or import them. This can be used for creating profiles that stretch across multiple machines. Just set the desired preferences on one machine, export them to a file, distribute this file, and import it.

Note: INSPECTOR EIFFEL *preferences are separate from the general EiffelStudio preferences. Pressing* Restore Defaults, Import "...", *or* Export "..." only *affects preferences for* INSPECTOR EIFFEL.

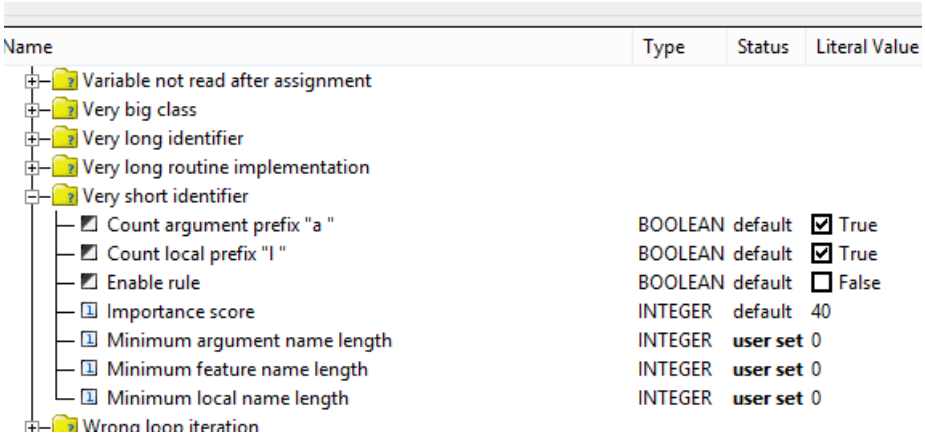

Figure 2.8: Rule-specific preferences.

Class Options

There are cases in which you might want to customize code analysis for *parts of your code* only. Inspector Eiffel provides a way to set options *per class*. You can exclude a class from being checked by certain rules. Also you can declare a class to be a *library* or a *non-library* class. All class-wide options for Inspector Eiffel are set in the *indexing clause* (after the `note` keyword).

Library and Non-Library Classes If the programmer uses the default values then a rule checks all classes. But a rule can be defined (hard-coded) not to check either *library* or *non-library* classes. How does Inspector Eiffel now know which classes are *library* classes and which classes are *non-library* classes? This is defined by the user. If, for a certain class, the user does *not* define anything then the class will be analyzed in every case. Only if the user declares a class to be a *library* class then this class will not be checked by a rule that has disabled checking library classes. The same goes for classes that are declared as *non-library*.

To declare a class to be a *library* class add `ca_library : "true"` to the (top or bottom) indexing clause.

To declare a class to be a *nonlibrary* class add `ca_library : "false"` to the (top or bottom) indexing clause.

Classes Ignored By Rules You can declare a class to be *ignored* by certain rules, which is equivalent to saying that some rules shall be *disabled* for a class.

To let a class be ignored by certain rules, add the `ca_ignoredby` tag to the (top or bottom) indexing clause. Then put all the relevant *rule IDs* separated by commas in the content. It may look like this:

```
ca_ignoredby : "CA005,CA092"
```

Chapter 3

Case Studies

For all of the case studies including the self-analysis we used the default settings (see appendix A) except for disabling rule number 20[1].

3.1 EiffelBase

EiffelBase is the base library that is by default included in every Eiffel system. It is a "library of fundamental structures and algorithms covering the basics of computing" [18]. *EiffelBase* contains classes such as `INTEGER`, `LINKED_LIST`, `ARRAY`, or `FUNCTION`. It has been released under an open-source license.

We now present the results of a code analysis of the whole EiffelBase library, which contains more than 300 classes. The analysis was performed on EiffelBase as of February 10, 2014.

3.1.1 Results Overview

436 warnings and 248 suggestions have been generated, making a total of 684 rule violations. EiffelBase has an average *warning density* of **1.33** warnings per class. Table 3.1 shows the frequencies of the rules that were violated the most.

[1] Rule #20 (*"Variable not read after assignment"*) is triggered in the case a variable is not read in the main feature body but is read in the rescue clause. This is very often the case in our case studies. Enabling this rule would lead to many false rule violations. The reason for the behavior of this rule is the following: Our control flow graph framework does not yet support exception handling. Hence, the rule, which uses the control flow graph functionality, ignores the rescue clause and assumes that the variable is not read at all.

No command-query separation (W #4)	161
Unneeded parentheses (S #23)	110
Many feature arguments (W #11)	103
Unneeded helper variable (S #85)	63
Creation procedure is exported (W #13)	51
Deeply nested `if` instructions (W #43)	40
High complexity of nested branches and loops (W #10)	31
Short-circuit `ifs` (S #28)	22

Table 3.1: Frequency table of rule violations in EiffelBase (top eight).

3.1.2 Notable Rule Violations

Code Structure and Code Size at Large

14 classes are considered very big and 3 routine implementations are considered very long by INSPECTOR EIFFEL. One of the big classes, `CHARACTER_PROPERTY`, contains 73 features and 342 instructions. The class file has 1863 lines. This class consists of "[h]elper functions to provide property of a Unicode character of type `CHARACTER_32`" (quoting from the class description). Indeed the class contains over a dozen helper functions. Most of the features though are attributes that are hard coded arrays with often more than 50 elements.

`CHARACTER_PROPERTY` also stands out due to the complexity of some features. The functions `to_lower`, `to_upper`, and `property` all have triggered both the "High NPATH" and the "High complexity of nested branches and loops" rules. To the reader of the code they look very complex indeed. The metrics of `property` for example, having 8 nested branches and loops and having an NPATH measure of 388316838, confirm this result.

Function Side Effects

There is an alarmingly high number of "No command-query separation" warnings. We should take them with a pinch of salt. It is obvious that in most or all of the cases this design is intentional. The rule is triggered for example when a function contains a procedure call. This must not necessarily mean that the command-query separation *principle* is violated.

The most affected class is `FILE`. A lot of functions contain a call to the `set_buffer` procedure, which "[r]esynchronizes information on [the] file". That seems appropriate and necessary. It is thus no cause for concern.

Redundancies and Possible Shortcuts

The code in Listing 3.1 is taken from class `LIST`. `cursor` is defined in `CURSOR_STRUCTURE`. It is of type `CURSOR`. Since `other` is of type `LIST`, also `other.cursor` must conform to `CURSOR`, and both two object tests for type `CURSOR` are redundant.

Listing 3.1: Extract from LIST.

```
1  if
2    attached {CURSOR} cursor as c1 and then
3    attached {CURSOR} other.cursor as c2
4  then
```

Listing 3.2: From {OBJECT_GRAPH_TRAVERSABLE}.internal_traverse.

```
1  if l_reflected_object.is_special then
2    if l_reflected_object.is_special_of_reference then
3      if attached {SPECIAL [detachable ANY]} l_object as l_sp then
4        from
5          i := 0
6          nb := l_sp.count
7        until
8          i = nb
9        loop
10         if l_reflected_object.is_special_copy_semantics_item (i)
               then
```

The rule "Two `if` instructions can be combined using short-circuit operator" was triggered 22 times. This code can certainly be simplified without risking any kind of drawbacks.

3.1.3 Proposals

Based on the outcome of our analysis we propose the following changes:

Extract Routines

Parts of the code in complex and very long routines should be extracted to separate routines. Listings 3.2 and 3.3 showcase relevant parts of features that each caused at least 3 rule violations. Starting at a certain depth of conditionals and loops, the "inner" code should be moved to new routines.

Listing 3.3: Small extract from {CHARACTER_PROPERTY}.property.

```
1  if (128 <= l_code) and (l_code <= 687) then
2    Result := property_table_1.item ((l_code - 128).to_integer_32)
3  else
4    if l_code <= 43002 then
5      if l_code <= 6829 then
6        if l_code <= 4968 then
7          if l_code <= 3439 then
8            if l_code <= 2142 then
9              if l_code <= 1805 then
10               if l_code >= 880 then
```

Listing 3.4: From ISE_EXCEPTION_MANAGER.

```
1 set_exception_data (code: INTEGER; new_obj: BOOLEAN;
2   signal_code: INTEGER; error_code: INTEGER; tag,
3   recipient, eclass: STRING; rf_routine, rf_class: STRING;
4   trace: STRING; line_number: INTEGER; is_invariant_entry: BOOLEAN)
```

Reduce Number of Arguments

Features with a high number of arguments should be redesigned. To reduce the number of arguments one can split the functionality into multiple features. Additionally, a default value can be defined for some arguments so the client only needs to set the value (calling a different feature) if it differs from the default. The most severe case in shown in Listing 3.4.

Use `and then` Operators

Nested `if` instructions according to rule #28 should be combined using the short-circuit `and then` operators.

Consistent Use of Parentheses

Not all of the 110 parentheses that were marked as unneeded must be removed. What we propose however is to use parentheses consistently in EiffelBase (or, even better, throughout all Eiffel libraries). An example of inconsistent use of parentheses is the expression

```
((Result.lower = lower) and (Result.upper = upper))
```

in line 244 of `ARRAY`. Yet line 256 of the same class is

```
elseif lower = other.lower and then upper = other.upper and then.
```

This line is missing two pairs of parentheses compared to line 244.

3.2 EiffelVision

The analysis was performed on EiffelVision as of February 10, 2014.

3.2.1 Results Overview

1074 warnings and 685 suggestions have been generated when *EiffelVision* was analyzed. EiffelVision consists of 745 classes, so the average *warning density* is **1.44** warnings per class. Table 3.2 shows the frequencies of the rules that were violated the most.

Many feature arguments (W #11)	360
No command-query separation (W #4)	336
Unneeded helper variable (S #85)	198
Deeply nested `if` instructions (W #43)	168
Unneeded parentheses (S #23)	160
Short-circuit `if`s (S #28)	93
`from-until` loop into `across` loop (S #24)	78

Table 3.2: Frequency table of rule violations in EiffelVision (top seven).

3.2.2 Notable Rule Violations

Many Feature Arguments

It is no coincidence that "Many feature arguments" is the most frequent rule violation for EiffelVision. Many of the correspondent features are used as agents and therefore require 4 or more arguments. Additionally there is a very large number of arguments representing coordinates. This is very typical and normal for a graphical user interface library. "x", "y", "width", and "height" (or similarly named) arguments are used very frequently throughout the library. Often, additional arguments are present, leading to an argument count that is certainly above the default threshold.

Function Side Effects

Many features triggered the "No command-query separation" rule, which essentially means that these functions are suspected to change the state of the object. We must investigate the cases further in order to determine whether they really change the external state of the object or whether they just call some internal features to optimize or cache something.

In the cases from EiffelVision almost all such violations are indeed not problematic. A very large amount of them is caused by functions that create *internal* variables when they are called for the first time. Then, many of the affected classes are descendants of `ITERABLE` and iterate over themselves inside a function. Some rule violations point to calls to procedures that update cached data if necessary. In a graphical user interface library this seems to be necessary more often than in other kinds of code.

We discovered the function `{WEL_WINDOW}.process_message` which contains procedure calls. In Listing 3.5 are the first few lines thereof. It resembles very much a window "message loop" that is used for example in the Windows API. It is actually used (in an indirect way) for the interface to the operating system.

A function where we could not identify the purpose of its result is the one in Listing 3.6. Such cases were very rare though.

Listing 3.5: Excerpt from {WEL_WINDOW}.process_message

```
1  window_process_message, process_message (hwnd: POINTER; msg:
       INTEGER;
2    wparam, lparam: POINTER): POINTER
3      -- Call the routine 'on_*' corresponding to the
4      -- message 'msg'.
5    require
6      exists: exists
7    local
8      l_message: detachable WEL_COMMAND_EXEC
9      l_commands: like commands
10   do
11     inspect msg
12       when Wm_mousemove then
13         on_mouse_move (wparam.to_integer_32,
14         x_position_from_lparam (lparam),
15         y_position_from_lparam (lparam))
16       when Wm_setcursor then
17         on_set_cursor (cwin_lo_word (lparam))
18       when Wm_windowposchanging then
19         on_wm_window_pos_changing (lparam)
```

Listing 3.6: A nontypical violation of "command-query separation".

```
1  show_disabled_close_button: BOOLEAN
2      -- Ensure 'disabled_close_button_shown' is 'True'.
3    do
4      disabled_close_button_shown := True
5    end
```

Listing 3.7: From {EV_EDITABLE_LIST}.is_vaild_text.

```
1 if (a_string.is_equal (item @ c) and not (index = r)) then
2   Result := False
3 elseif (not empty_column_values) and (a_string.is_empty) then
4   Result := False
5 end
```

Listing 3.8: From EV_POSTSCRIPT_DRAWABLE_IMP.

```
1  draw_point (x, y: INTEGER)
2      -- Draw point at ('x', 'y').
3    do
4      translate_to (x, ( - y))
5      add_ps ("newpath")
6      draw_arc_ps (1, 0, 360)
7      add_ps ("closepath")
8      add_ps ("fill")
9      add_ps ("stroke")
10     translate_to (-x, y)
11   end
```

Conditionals, Loops, and Code Style

Whereas the deeply nested `if` instructions can be left untouched (in most of the cases the code is still well-readable), there are 93 cases where at least one `if` instruction can be avoided by using the `and then` operator.

Most of the unneeded parentheses that Inspector Eiffel found in EiffelVision are arguable. In Listing 3.7 Inspector Eiffel suggests to remove the outer parentheses in line 1 and to remove the right pair of parentheses in line 3.

The next example in Listing 3.8 has unneeded parentheses (according to our definition) in line 4 (`( - y)`). Notice the inconsistency (line 10) within the same routine.

Also notable regarding coding style are the 78 cases where a `from-until` loop can be rewritten as an `across` loop. For some reason this number is significantly higher compared to EiffelBase (where there were 17 such rule violations).

3.2.3 Proposals

Based on our analysis we propose the following changes to *EiffelVision*:

Simplify `ifs` Using `and then`

Nested `if` instructions should be combined using the short-circuit `and then` operators. This will make all 93 violations of rule #28 disappear.

No command-query separation (W #4)	353
Deeply nested `if` instructions (W #43)	332
Many feature arguments (W #11)	205
High NPATH (W #34)	147
High complexity of nested branches and loops (W #10)	138
Very long routine implementation (W #32)	70
Creation procedure is exported (W #13)	68

Table 3.3: Frequency table of warnings in the `compiler` cluster (top seven).

Consistent Style For Loops

EiffelVision does not use `across` loops to iterate over an `ITERABLE` instance.[2] Only conventional `from-until` loops are used. There are 78 cases which indicate a `from-until` loop that can be transformed into an `across` loop. We propose to consider using the more up-to-date `across` loop whenever it is possible.

Use Parentheses Consistently

Parentheses should be used in a consistent manner throughout EiffelVision—or, better, throughout all Eiffel libraries.

Revise Feature Arguments

Despite the special nature of a graphical library, in which coordinates are used in many places, the features that have many arguments (there are 360 of them) should be revised. Having less arguments makes features easier to use for clients.

3.3 EiffelStudio

We analyzed a small exemplary part from the huge EiffelStudio source code: the cluster `compiler` from the library `compiler_kernel`. It has 683 classes. Due to this large number of classes we first show a statistical overview of the results and then pick a single class for a detailed discussion. The analysis was performed on the source code as of March 7, 2014.

3.3.1 Results Overview

Table 3.3 shows the top frequencies of warnings, of which 1382 were generated. The 951 suggestions are not shown. The `compiler` cluster has an average *warning density* of **2.02** warnings per class, which is considerably higher than the values of *EiffelBase* and *EiffelVision*.

[2] `across` loops have been introduced into Eiffel only in 2010 [12].

	NPATH	#43	#10	#32
fill_parents	4482	x	x	x
update_generic_features	1470	x	x	–
update_anchors	540	x	–	–
process_skeleton	–	x	x	–
is_fully_deferred	–	x	x	–
check_validity	24961	–	–	–

Table 3.4: Features with high complexity.

3.3.2 CLASS_C In Detail

CLASS_C represents a compiled class. The Eiffel language alone defines many properties of a class, many of which must be represented in CLASS_C—a class for a "class". Many features actually deal with the relation to other classes. It is these classes that generated many of the warnings.

We observe a conspicuously large number of warnings related to complexity. Table 3.4 lists features that caused significant rule violations regarding routine complexity. The NPATH measure is stated only if it caused a rule violation, i. e., if it is above the threshold of 200.

We think that one should attempt to extract functionality from the reported features to separate (new) features or to other (new) classes.

3.4 Self-Analysis

Analyzing our own code is a very important and a very interesting case study. As you will see Inspector Eiffel actually found a few issues in its own code, even though—as compared to the case studies above—we had the advantage of knowing how to code in order to avoid rule violations. The analysis was performed on the Inspector Eiffel code as of February 12, 2014.

3.4.1 Results Overview

The self-analysis has generated 39 warnings and 8 suggestions. The Inspector Eiffel framework contains 76 classes, making it an average *warning density* of **.51** warnings per class. Table 3.5 shows the frequencies of the violated rules.

3.4.2 Commentary

In the following we will comment on the most significant issues in our code. Sometimes there is a clear reason for rule violations, which we are going to explain. Other issues are just shown as they are. We have left them deliberately open for discussion.

Deeply nested `if` instructions (W #43)	21
Unneeded helper variable (S #85)	6
No command-query separation (W #4)	5
Very big class (W #33)	4
Many feature arguments (W #11)	4
High NPATH (W #34)	2
High complexity of nested branches and loops (W #10)	2
Missing `is_equal` redefinition (W #82)	1
`from-until` loop into `across` loop (S #24)	1
Unneeded parentheses (S #23)	1

Table 3.5: Frequency table of rule violations in the code analysis framework.

Deeply Nested `if` Instructions

The large number thereof is mainly due to the manner in which many static analyses on the abstract syntax tree work. With 12 rule violations located in `CA_PRETTY_PRINTER`, a class that we took from an external source, only *9* violations affect our own code. Often in our analyses we start at an AST node. From there we follow the object graph. Usually this includes some conditionals such as object tests. Despite some nested conditionals the code remains well-readable in our opinion. Listing 3.9 shows an example.

Unneeded Helper Variable

The 6 suggestions to remove helper variables essentially concern coding style. In all the cases we decided to leave it as it is. Leaving the helper variable there makes the code easier to read. With "unneeded helper variable" suggestions one should always check the code context before changing the code. Listings 3.10, 3.11, and 3.12 show extracts from the Inspector Eiffel source.

No Command-Query Separation

One of the rule violations is caused by the function
`{CA_RULE_VIOLATION}.csv_line`
which calls
`{CA_RULE_VIOLATION}.format_violation_description (a_tf: TEXT_FORMATTER)`.
`format_violation_description` is syntactically a procedure but acts as a function because `a_tf` is an in-out argument. So this rule violation is a false positive. The procedure call does *not* change the state of the object (as most of the procedure calls do). This shows a weakness of the rule implementation.

All the other violations of command-query separation affect the class `CA_VARIABLE_NOT_READ_RULE`. In its function
`visit_edge (a_from, a_to: attached CA_CFG_BASIC_BLOCK): BOOLEAN`
from `CA_CFG_ITERATOR` lies the cause of the violations. Processing the edge and using the result to decide whether to carry on iterating (see Section 4.4.4 for

Listing 3.9: Extract from CA_SELF_COMPARISON_RULE.

```
pre_process_loop (a_loop: LOOP_AS)
    -- Checking a loop 'a_loop' for self-comparisons needs more work. If the until expression
    -- is a self-comparison that does not compare for equality then the loop will
    -- not terminate, which is more severe consequence compared to other self-comparisons.
  local
    l_viol: CA_RULE_VIOLATION
  do
    if attached {BINARY_AS} a_loop.stop as l_bin then
      analyze_self (l_bin)
      if is_self then
        create l_viol.make_with_rule (Current)
        l_viol.set_location (a_loop.stop.start_location)
        l_viol.long_description_info.extend (self_name)
        if not attached {BIN_EQ_AS} l_bin then
            -- It is only a dangerous loop stop condition if we do not have
            -- an equality comparison.
          l_viol.long_description_info.extend ("loop_stop")
        end
        violations.extend (l_viol)
        in_loop := True
      end
    end
  end
```

Listing 3.10: From CA_NPATH_RULE.

```
inner_npath := npath_stack.item + 1
npath_stack.remove
outer_npath := npath_stack.item
npath_stack.replace (inner_npath * outer_npath)
```

Listing 3.11: From CA_VARIABLE_NOT_READ_RULE.

```
l_from := a_from.label
l_to := a_to.label

Result := node_union (l_from, l_to)
```

Listing 3.12: From CA_UNNEEDED_HELPER_VARIABLE_RULE.

```
l_used := locals_usage [l_id]
locals_usage.force (l_used + 1, l_id)
```

details) is convenient although it could also be done by writing to an attribute instead of using the result.

Very Big Class

Four classes were reported as very big:

`CA_ALL_RULES_CHECKER` The vast majority of its 99 features are visitors and `ACTION_SEQUENCE` attributes, both of which are used to process the various types of abstract syntax tree nodes during analysis.

`CA_NAMES` **and** `CA_MESSAGES` These classes with approximately 100 features each contain all the localized strings.

`CA_PRETTY_PRINTER` This class was taken as it was from an external source and will therefore not be discussed here.

Many Feature Arguments

All the reported features have 4 or 5 arguments, which may be not ideal but is acceptable. (The default threshold is at 4 arguments.) Two of this features are creation procedures, the other two are `record_node_type` and `type_of_node` from `CA_AST_TYPE_RECORDER`. This class depends heavily on the interface to the Eiffel compiler.

High NPATH

Both features with a high NPATH measure belong to `CA_PRETTY_PRINTER` which, as explained before, we will not discuss.

High Complexity of Nested Branches and Loops

These rule violations are closely related to those about "deeply nested `if` instructions". Again we follow the object graph which requires object tests, loops, and other conditionals. Listing 3.13 is taken from the implementation of rule #13: *creation procedure is exported.*

Listing 3.13: Extract from CA_CREATION_PROC_EXPORTED_RULE.

```
process_feature_clause (a_clause: FEATURE_CLAUSE_AS)
    -- Checks 'a_clause' for features that are creation procedures.
  local
    l_feature: FEATURE_AS
    l_exported: BOOLEAN
    l_viol: CA_RULE_VIOLATION
  do
    if creation_procedures /= Void then
      across creation_procedures as ic loop
        l_feature := a_clause.feature_with_name (ic.item.
            internal_name.name_id)
        if l_feature /= Void then
          if attached a_clause.clients as l_clients then
            across l_clients.clients as l_class_list loop
              if not l_class_list.item.name_32.is_equal ("NONE")
                  then
                  -- The feature is exported to something.
                l_exported := True
              end
            end
          else
              -- No clients are defined. It means that the feature
                  is exported to {ANY}.
            l_exported := True
          end

          if l_exported then
            create l_viol.make_with_rule (Current)
            l_viol.set_location (a_clause.start_location)
            l_viol.long_description_info.extend (l_feature.
                feature_name.name_32)
            violations.extend (l_viol)
          end
        end
      end
    end
  end
```

Chapter 4

Implementation

The code for Inspector Eiffel is located at three different places in the *EiffelStudio* source:

1. The framework—by far the largest part, with the rule checking, the rules, the control flow graph functionality, and more—is represented as a *library*;
2. The graphical user interface can be found in the `interface` cluster of *EiffelStudio*;
3. The command-line interface for Inspector Eiffel is a single class in the `tty` cluster of *EiffelStudio*.

4.1 Library Implementation

The whole Inspector Eiffel framework is located in the library `code_analysis`.

4.1.1 Class Relations

The diagram in Figure 4.1 shows an overview of the relations between the classes of the Inspector Eiffel framework. All classes are located in the `code_analysis` library except for:

- `CLASS_C` (EiffelStudio),
- `ROTA_TIMED_TASK_I` (`ecosystem` cluster),
- `EWB_CODE_ANALYSIS` (command-line interface),
- `ES_CODE_ANALYSIS_BENCH_HELPER` (GUI).

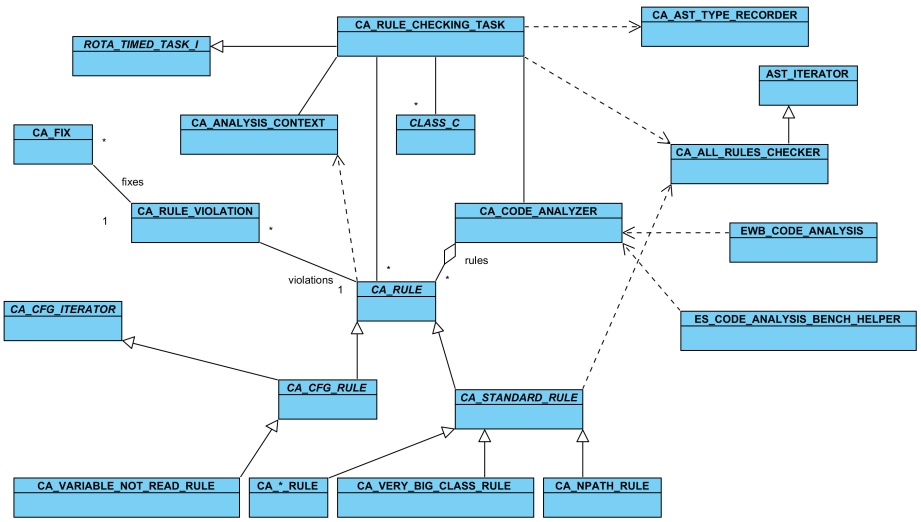

Figure 4.1: The most relevant classes of the INSPECTOR EIFFEL framework.

4.1.2 Interface

In this section it is explained from a client view how to use INSPECTOR EIFFEL. The INSPECTOR EIFFEL code analyzer is represented by the class `CA_CODE_ANALYZER`, so a client must have or access an instance of this class. Before the analyzer can be launched all the classes that shall be analyzed must be added using one of the following features. If you use more than one of these commands then the added classes from all commands will be conjoined.

`{CA_CODE_ANALYZER}.add_whole_system` — Adds all the classes that are part of the current system. Classes of referenced libraries will not be added. So, for example, if your system consists of the classes `MY_MAIN`, `MY_BOX`, and `MY_ITEM` then these three classes will be added to the list of classes to be analyzed.

`.add_class (a_class: attached CONF_CLASS)` — Adds a single class.

`.add_classes (a_classes: attached ITERABLE [attached CONF_CLASS])` — Adds a list of classes.

`.add_cluster (a_cluster: attached CLUSTER_I)` — Adds all classes of a cluster (and all the classes of the sub-clusters recursively).

`.add_group (a_group: attached CONF_GROUP)` — Adds all classes of a configuration group. An example of a configuration group is a *library*.

There are other features which can be called before starting to analyze:

`{CA_CODE_ANALYZER}.clear_classes_to_analyze` — Removes all classes that have been added to the list of classes to analyze.

`.add_completed_action (a_action: attached PROCEDURE [ANY, TUPLE [ITERABLE [TUPLE [detachable EXCEPTION, CLASS_C]]]])` — Adds `a_action` to the list of procedures that will be called when analysis has completed. The procedures have one argument, a list of exceptions (with the corresponding class).

Listing 4.1: From {CA_CODE_ANALYZER}.analyze.

```
1 create l_task.make (l_rules_checker, l_rules_to_check,
      classes_to_analyze, agent analysis_completed)
2 l_task.set_output_actions (output_actions)
3 rota.run_task (l_task)
```

In the case an exception is thrown during analysis the exception is caught by the code analyzer and is added to this list. In the graphical user interface such exceptions would show up as errors at the top of the list of rule violations.

`.add_output_action (a_action: attached PROCEDURE [ANY, TUPLE [READABLE_STRING_GENERAL]])` — Adds `a_action` to the procedures that are called for outputting the status. The final results (rule violations) are not given to these procedures. These output actions are used by the command-line mode and by the status bar in the GUI.

`.is_rule_checkable (a_rule: attached CA_RULE): BOOLEAN` — Tells whether `a_rule` will be checked based on the current preferences and based on the current checking scope (whole system or custom set of classes).

Then, to start analyzing simply call `{CA_CODE_ANALYZER}.analyze`.

4.1.3 Rule Checking

In the GUI we want to be able to continue to work while Inspector Eiffel is running. Analyzing larger sets of classes (such as whole libraries) can take from several seconds to several minutes. For this reason the code analyzer uses an *asynchronous* task, `CA_RULE_CHECKING_TASK`. In `{CA_CODE_ANALYZER}.analyze` this task (`l_task`) is invoked as shown in Listing 4.1.

`CA_RULE_CHECKING_TASK` essentially carries out the whole analysis. Like all other subclasses of `ROTA_TASK_I` this class executes a series of *steps* between which the user interface gets some time to process its events. In `CA_RULE_CHECKING_TASK` each step analyses one class. This means that a class is checked for violations by *all* the rules. This is done by the code from Listing 4.2.

`type_recorder` is of type `CA_AST_TYPE_RECORDER`. It uses a functionality of the Eiffel compiler to determine the type of some AST nodes in the current class. The AST itself (as provided by the Eiffel compiler) does not contain complete type information. `context` has type `CA_ANALYSIS_CONTEXT` and contains any side-information such as the previously mentioned types and the current class. The rules were given this context before so that they can access it when needed.

The `across` loop only checks *control flow graph rules*. All the *standard* rules are checked by the line `rules_checker.run_on_class (classes.item)`. `rules_checker` has type `CA_ALL_RULES_CHECKER`. This is the class where each rule must register the AST nodes the rule visits. `run_on_class` iterates over the AST and calls all the actions that were registered by the standard rules. So this is the way all rules are used to check the current class. `step` is executed repeatedly

Listing 4.2: From CA_RULE_CHECKING_TASK.

```
step
    -- <Precursor>
  do
    if has_next_step then
        -- Gather type information
      type_recorder.clear
      type_recorder.analyze_class (classes.item)
      context.set_node_types (type_recorder.node_types)
      context.set_checking_class (classes.item)

      across rules as l_rules loop
          -- If rule is non-standard then it will not be checked by
               l_rules_checker.
          -- We will have the rule check the current class here:
        if
          l_rules.item.is_enabled.value
          and then attached {CA_CFG_RULE} l_rules.item as
              l_cfg_rule
        then
          l_cfg_rule.check_class (classes.item)
        end
      end

        -- Status output.
      if output_actions /= Void then
        output_actions.call ([ca_messages.analyzing_class (classes.
            item.name)])
      end

      rules_checker.run_on_class (classes.item)

      classes.forth
      has_next_step := not classes.after
      if not has_next_step then
        completed_action.call ([exceptions])
      end
    end
  rescue
      -- Instant error output.
    if output_actions /= Void then
      output_actions.call ([ca_messages.error_on_class (classes.
          item.name)])
    end
    exceptions.extend ([exception_manager.last_exception, classes.
        item])
      -- Jump to the next class.
    classes.forth
    has_next_step := not classes.after
    if not has_next_step then
      completed_action.call ([exceptions])
    end
    retry
  end
```

Listing 4.3: "Pre" and "post" actions.

```
1  if_pre_actions, if_post_actions: ACTION_SEQUENCE [TUPLE [IF_AS]]
2
3  add_if_post_action (a_action: attached PROCEDURE [ANY, TUPLE [IF_AS
       ]])
4    do
5      if_post_actions.extend (a_action)
6    end
7
8  -- And similar for all other relevant AST nodes...
```

Listing 4.4: An AST visitor routine.

```
1  process_if_as (a_if: IF_AS)
2    do
3      if_pre_actions.call ([a_if])
4      Precursor (a_if)
5      if_post_actions.call ([a_if])
6    end
7
8  -- And similar for all other relevant AST nodes...
```

until there are no more classes left to analyze.

In the `rescue` clause all possible exceptions are caught and recorded. In case of such an exception we proceed to the next class.

Checking *Standard* Rules

The relatively large class `CA_ALL_RULES_CHECKER` is responsible for checking *standard rules*. It does this in a straightforward way. It is a subclass of `AST_ITERATOR`, a realization of a visitor[1] on the AST.

Rules can register their actions with `CA_ALL_RULES_CHECKER` by calling a procedure like
`add_bin_lt_pre_action(a_action: attached PROCEDURE[ANY,TUPLE[BIN_LT_AS]])`
or
`add_if_post_action (a_action: attached PROCEDURE [ANY, TUPLE [IF_AS]])`.
These "pre" and "post" actions exist for many other types of AST nodes as well. All the registered actions are stored in `ACTION_SEQUENCE` variables, as shown in Listing 4.3. The corresponding visitor procedures are redefined. This is done as shown in Listing 4.4. Since the actual iteration over the AST is done in the ancestor we need only very little code to analyze a class.

Listing 4.5 shows code that analyzes a class with respect to all active *standard* rules. `class_pre_actions` and `class_post_actions` are action sequences that are identical to those for the AST nodes. `process_class_as`, which is implemented in `AST_ITERATOR` will recursively visit all relevant AST nodes and execute their action sequences.

[1] [4, p. 331ff.] discusses visitors and mentions their use in program analysis.

Listing 4.5: Analyzing a class in regard to standard rules.

```
1  feature {CA_RULE_CHECKING_TASK} -- Execution Commands
2
3    run_on_class (a_class_to_check: CLASS_C)
4        -- Check all rules that have added their agents.
5      local
6        l_ast: CLASS_AS
7      do
8        last_run_successful := False
9        l_ast := a_class_to_check.ast
10       class_pre_actions.call ([l_ast])
11       process_class_as (l_ast)
12       class_post_actions.call ([l_ast])
13       last_run_successful := True
14     end
```

4.2 Example: Rule #71: *Self-Comparison*

We will go through the implementation of rule #71 (*Self-comparison*) in detail.

The heart of this implementation lies in the feature `analyze_self`, which is shown in Listing 4.6. There it is tested whether a binary expression is a self-comparison. `is_self`, a `BOOLEAN` attribute, is set to `True` if and only if the argument is a comparison between two identical variables.

Both sides of the comparison, `a_bin.left` and `a_bin.right`, are tested for having the types that indicate that they are variable or feature accesses. If the tests succeed then `is_self` is set according to the equality of the two feature names. Then the name is stored in an internal attribute.

`analyze_self` is used in `process_comparison` (shown in Listing 4.7), which creates a rule violation if a self-comparison was detected.

First we check that we are not dealing with a loop condition. Self-comparisons in loop conditions are more dangerous and need special treatment (see Listing 4.9). For the rule violation, we set the location to the start location of the binary comparison. We add the variable or feature name to the violation.

Different kinds of comparisons also have different types in the AST. That is why in an AST iterator they are processed independently. Thus, we need to delegate to `process_comparison` in each of the actions that are called when processing a comparison. Listing 4.8 shows the visitors of the "binary" nodes of the AST.

In the case that a loop condition is a self-comparison, the loop is either never entered or it is never exited. Never exiting a loop is more severe; never entering only arises when it is an equality comparison (`=` or `~`). For this reason we analyze loop conditions separately. This is shown in Listing 4.9. If we find such a violation we set `in_loop` to `True` so that any further self-comparisons are ignored until we have left the loop.

`format_violation_description`, which is declared in `CA_RULE` as `deferred`, must be implemented (shown in Listing 4.10). Here, together with a prede-

Listing 4.6: {CA_SELF_COMPARISON_RULE}.analyze_self

```
1  analyze_self (a_bin: attached BINARY_AS)
2      -- Is 'a_bin' a self-comparison?
3    do
4      is_self := False
5
6      if
7        attached {EXPR_CALL_AS} a_bin.left as l_e1
8        and then attached {ACCESS_ID_AS} l_e1.call as l_l
9        and then attached {EXPR_CALL_AS} a_bin.right as l_e2
10       and then attached {ACCESS_ID_AS} l_e2.call as l_r
11     then
12       is_self := l_l.feature_name.is_equal (l_r.feature_name)
13       self_name := l_l.access_name_32
14     end
15   end
16
17 is_self: BOOLEAN
18     -- Is 'a_bin' from last call to 'analyze_self' a self-
           comparison?
19
20 self_name: detachable STRING_32
21     -- Name of the self-compared variable.
```

Listing 4.7: {CA_SELF_COMPARISON_RULE}.process_comparison

```
1  process_comparison (a_comparison: BINARY_AS)
2      -- Checks 'a_comparison' for rule violations.
3    local
4      l_viol: CA_RULE_VIOLATION
5    do
6      if not in_loop then
7        analyze_self (a_comparison)
8        if is_self then
9          create l_viol.make_with_rule (Current)
10         l_viol.set_location (a_comparison.start_location)
11         l_viol.long_description_info.extend (self_name)
12         violations.extend (l_viol)
13       end
14     end
15   end
```

Listing 4.8: Processing "binary" nodes in `CA_SELF_COMPARISON_RULE`

```
process_bin_eq (a_bin_eq: BIN_EQ_AS)
  do
    process_comparison (a_bin_eq)
  end

process_bin_ge (a_bin_ge: BIN_GE_AS)
  do
    process_comparison (a_bin_ge)
  end

process_bin_gt (a_bin_gt: BIN_GT_AS)
  do
    process_comparison (a_bin_gt)
  end

process_bin_le (a_bin_le: BIN_LE_AS)
  do
    process_comparison (a_bin_le)
  end

process_bin_lt (a_bin_lt: BIN_LT_AS)
  do
    process_comparison (a_bin_lt)
  end
```

Listing 4.9: `{CA_SELF_COMPARISON_RULE}.pre_process_loop`

```
pre_process_loop (a_loop: LOOP_AS)
    -- Checking a loop 'a_loop' for self-comparisons needs more
        work. If the until expression
    -- is a self-comparison that does not compare for equality then
        the loop will
    -- not terminate, which is more severe consequence compared to
        other self-comparisons.
  local
    l_viol: CA_RULE_VIOLATION
  do
    if attached {BINARY_AS} a_loop.stop as l_bin then
    analyze_self (l_bin)
    if is_self then
      create l_viol.make_with_rule (Current)
      l_viol.set_location (a_loop.stop.start_location)
      l_viol.long_description_info.extend (self_name)
      if not attached {BIN_EQ_AS} l_bin then
          -- It is only a dangerous loop stop condition if we do
              not have
          -- an equality comparison.
        l_viol.long_description_info.extend ("loop_stop")
      end
      violations.extend (l_viol)
      in_loop := True
    end
  end
end
```

Listing 4.10: {CA_SELF_COMPARISON_RULE}.format_violation_description

```
format_violation_description (a_violation: attached
    CA_RULE_VIOLATION; a_formatter: attached TEXT_FORMATTER)
  local
    l_info: LINKED_LIST [ANY]
  do
    l_info := a_violation.long_description_info
    a_formatter.add ("'")
    if l_info.count >= 1 and then attached {STRING_32} l_info.first
        as l_name then
      a_formatter.add_local (l_name)
    end
    a_formatter.add (ca_messages.self_comparison_violation_1)

    l_info.compare_objects
    if l_info.has ("loop_stop") then
        -- Dangerous loop stop condition.
      a_formatter.add (ca_messages.self_comparison_violation_2)
    end
  end
```

Listing 4.11: Properties of CA_SELF_COMPARISON_RULE

```
title: STRING_32
  do
    Result := ca_names.self_comparison_title
 end

id: STRING_32 = "CA071"
    -- <Precursor>

description: STRING_32
  do
    Result :=  ca_names.self_comparison_description
  end
```

fined localized text, we mention the name of the self-compared variable. If the self-comparison is located in a loop stop condition we add an additional warning text. Then we must implement the usual properties, as shown in Listing 4.11.

Finally, in the initialization (shown in Listing 4.12) we use the default settings, which can be set by calling `{CA_RULE}.make_with_defaults`. To the default severity score we assign a custom value. In `register_actions` we must add all the agents for processing the loop and comparison nodes of the AST.

4.3 Example: Rule #2: *Unused Argument*

The *unused argument* rule processes the *feature*, *body*, *access id*, and *converted expression* AST nodes. The feature node is stored for the description and for ignoring *deferred* features. The body node is used to retrieve the arguments. The *access id* and *converted expression* nodes may represent used arguments,

Listing 4.12: Initialization in `CA_SELF_COMPARISON_RULE`

```
1  feature {NONE} -- Initialization
2
3    make
4        -- Initialization.
5      do
6        make_with_defaults
7        default_severity_score := 70
8      end
9
10 feature {NONE} -- Activation
11
12   register_actions (a_checker: attached CA_ALL_RULES_CHECKER)
13     do
14       a_checker.add_bin_eq_pre_action (agent process_bin_eq)
15       a_checker.add_bin_ge_pre_action (agent process_bin_ge)
16       a_checker.add_bin_gt_pre_action (agent process_bin_gt)
17       a_checker.add_bin_le_pre_action (agent process_bin_le)
18       a_checker.add_bin_lt_pre_action (agent process_bin_lt)
19       a_checker.add_loop_pre_action (agent pre_process_loop)
20       a_checker.add_loop_post_action (agent post_process_loop)
21     end
```

Listing 4.13: `{CA_UNUSED_ARGUMENT_RULE}.register_actions`

```
1  feature {NONE} -- Activation
2
3    register_actions (a_checker: attached CA_ALL_RULES_CHECKER)
4      do
5        a_checker.add_feature_pre_action (agent process_feature)
6        a_checker.add_body_pre_action (agent process_body)
7        a_checker.add_body_post_action (agent post_process_body)
8        a_checker.add_access_id_pre_action (agent process_access_id)
9        a_checker.add_converted_expr_pre_action (agent
             process_converted_expr)
10     end
```

so the nodes are used to mark arguments as read. We register the "pre" actions for all the AST nodes as well as the "post" action for the *body* node in `register_actions`, shown in Listing 4.13.

On processing a feature we store the feature instance, which will be used later. This is shown in Listing 4.14. Listing 4.15 shows the "pre" action for *body* nodes. Before processing the body of a feature we store a list of all the argument names. This is however only done if the feature is a routine, it has arguments, and it is not external. In the code we need two nested loops since the arguments are grouped by type. For example, two consecutive `STRING` arguments as in `feature print(first, second: STRING)` are contained in one entry of `{BODY_AS}.arguments`.

Both the nodes `ACCESS_ID_AS` and `CONVERTED_EXPR_AS` may represent used arguments. `ACCESS_ID_AS` is a usual variable usage, while `CONVERTED_EXPR_AS` stands for an argument used in inline C code (the dollar sign syntax: `$arg`). In both routines `check_arguments` is called eventually, which updates the internal

Listing 4.14: {CA_UNUSED_ARGUMENT_RULE}.process_feature

```
process_feature (a_feature_as: FEATURE_AS)
    -- Sets the current feature.
  do
    current_feature := a_feature_as
  end
```

Listing 4.15: {CA_UNUSED_ARGUMENT_RULE}.process_body

```
process_body (a_body_as: BODY_AS)
    -- Retrieves the arguments from 'a_body_as'.
  local
    j: INTEGER
  do
    has_arguments := (a_body_as.arguments /= Void)
    create args_used.make (0)
    n_arguments := 0
    if
      attached a_body_as.as_routine as l_rout
      and then has_arguments
      and then not l_rout.is_external
    then
      routine_body := a_body_as
      create arg_names.make (0)
      across a_body_as.arguments as l_args loop
        from
          j := 1
        until
          j > l_args.item.id_list.count
        loop
          arg_names.extend (l_args.item.item_name (j))
          args_used.extend (False)
          n_arguments := n_arguments + 1
          j := j + 1
        end
      end
    end
  end

has_arguments: BOOLEAN
    -- Does current feature have arguments?

current_feature: FEATURE_AS
    -- Currently checked feature.

routine_body: BODY_AS
    -- Current routine body.

n_arguments: INTEGER
    -- # arguments for current routine.

arg_names: ARRAYED_LIST [STRING_32]
    -- Argument names of current routine.

args_used: ARRAYED_LIST [BOOLEAN]
    -- Which argument has been used?
```

Listing 4.16: Checking for unused arguments.

```
process_access_id (a_aid: ACCESS_ID_AS)
    -- Checks if 'a_aid' is an argument.
  do
    check_arguments (a_aid.feature_name.name_32)
  end

process_converted_expr (a_conv: CONVERTED_EXPR_AS)
    -- Checks if 'a_conv' is an argument used in the
    -- form '$arg'.
  local
    j: INTEGER
  do
    if
      attached {ADDRESS_AS} a_conv.expr as l_address
      and then attached {FEAT_NAME_ID_AS} l_address.feature_name as
          l_id
    then
      check_arguments (l_id.feature_name.name_32)
    end
  end

check_arguments (a_var_name: attached STRING_32)
    -- Mark an argument as used if it corresponds to 'a_aid'.
  local
    j: INTEGER
  do
    from
      j := 1
    until
      j > n_arguments
    loop
      if not args_used [j] and then arg_names [j].is_equal (
          a_var_name) then
        args_used [j] := True
      end
      j := j + 1
    end
  end
```

data structures of our rule class. This is shown in Listing 4.16.

`post_process_body` (shown in Listing 4.17) finally checks if there exist unused arguments. If this is the case then all the relevant variable names are stored in the rule violation. Also, the feature is stored (for the feature name). The location of the violation is set to the start of the routine body. No rule violation is issued if the feature is deferred.

All the information that was stored in the rule violation is used for the formatted description, shown in Listing 4.18.

Listing 4.17: {CA_UNUSED_ARGUMENT_RULE}.post_process_body

```
1  post_process_body (a_body: BODY_AS)
2      -- Adds a violation if the feature contains unused arguments.
3    local
4      l_violation: CA_RULE_VIOLATION
5      j: INTEGER
6    do
7      if
8        a_body.content /= Void
9        and then not current_feature.is_deferred
10       and then has_arguments
11       and then args_used.has (False)
12     then
13       create l_violation.make_with_rule (Current)
14       l_violation.set_location (routine_body.start_location)
15       l_violation.long_description_info.extend (current_feature)
16       from
17         j := 1
18       until
19         j > n_arguments
20       loop
21         if not args_used.at (j) then
22           l_violation.long_description_info.extend (arg_names.at (j
                 ))
23         end
24         j := j + 1
25       end
26       violations.extend (l_violation)
27     end
28   end
```

Listing 4.18: {CA_UNUSED_ARGUMENT_RULE}.formatted_description

```
1  format_violation_description (a_violation: attached
       CA_RULE_VIOLATION; a_formatter: attached TEXT_FORMATTER)
2    local
3      j: INTEGER
4    do
5      a_formatter.add (ca_messages.unused_argument_violation_1)
6      from
7        j := 2
8      until
9        j > a_violation.long_description_info.count
10     loop
11       if j > 2 then a_formatter.add (", ") end
12       a_formatter.add ("'")
13       if attached {STRING_32} a_violation.long_description_info.at
             (j) as l_arg then
14         a_formatter.add_local (l_arg)
15       end
16       a_formatter.add ("'")
17       j := j + 1
18     end
19     a_formatter.add (ca_messages.unused_argument_violation_2)
20     if attached {FEATURE_AS} a_violation.long_description_info.
           first as l_feature then
21       a_formatter.add_feature_name (l_feature.feature_name.name_32,
             a_violation.affected_class)
22     end
23     a_formatter.add (ca_messages.unused_argument_violation_3)
24   end
```

4.4 Adding New Rules

The Inspector Eiffel framework was designed with regard to the fact that adding new rules should be as simple and as fast as possible. Looking at the set of rules that we implemented, nearly all of them have an implementation of less than 200 lines of code. Many of them use even less than 100 lines of code. Rules that search the code for certain patterns (this applies to the vast majority of rules) are particularly simple to implement.

This section is about implementing a rule in the form of a class. After writing such a class the rule must be added to the list of rules. This list is populated in {CA_CODE_ANALYZER}.make. There, a line like

```
rules.extend (create {YOUR_RULE}.make)
```

must be added, where YOUR_RULE must be replaced by the name of your rule class and the creation procedure make must be adapted if necessary.

4.4.1 Standard Rules

All rules must conform to CA_RULE. The class you implement for a rule is on one hand responsible for checking the rule and contains metadata about the rule (i. e., title, description) on the other hand. Rules must moreover conform to either CA_STANDARD_RULE or CA_CFG_RULE, both of which are subtypes of CA_RULE. A large number of possible rules are standard rules, no matter whether they are trivial or more complicated.

All *standard rules* are checked by iterating over the AST of the class code. The developer who adds a new rule can very well ignore the details thereof. He needs to know however which AST nodes his rule needs to process. For each type of AST node you need to add an agent so your routine will be called during the iteration on the AST.

To start implementing a rule one has basically two possibilities.

1. Starting from scratch, implementing all deferred features of CA_STANDARD_RULE;
2. Using a template such as Listing 4.19.

Listing 4.19: Standard rule template.

```
1  class
2    CA_YOUR_RULE
3
4  inherit
5    CA_STANDARD_RULE
6
7  create
8    make
9
10 feature {NONE} -- Initialization
11
12   make (a_pref_manager: attached PREFERENCE_MANAGER)
```

```
13        -- Initialization for `Current'.
14      do
15        make_with_defaults
16          -- This initializes the attributes to their default values:
17          -- Severity = warning
18          -- Default Severity Score = 50 (`severity score' can be
                changed by user)
19          -- Rule enabled by default = True (`Rule enabled' can be
                changed by user)
20          -- Only for system wide checks = False
21          -- Checks library classes = True
22          -- Checks nonlibrary classes = True
23
24          initialize_options (a_pref_manager)
25
26          -- TODO: Add your initialization here.
27      end
28
29    initialize_options (a_pref_manager: attached PREFERENCE_MANAGER)
30        -- Initializes the rule preferences.
31      local
32        l_factory: BASIC_PREFERENCE_FACTORY
33      do
34        create l_factory
35
36          -- TODO: Add the initialization of your custom preferences
                here.
37          -- Example:
38 --       threshold := l_factory.new_integer_preference_value (
        a_pref_manager,
39 --         preference_namespace + "Threshold",
40 --         30)  -- default value
41 --       min_local_name_length.set_default_value ("30") -- default
        value, too
42 --       min_local_name_length.set_validation_agent (agent
        is_integer_string_within_bounds (?, 1, 1_000_000))
43      end
44
45  feature {NONE} -- Activation
46
47    register_actions (a_checker: attached CA_ALL_RULES_CHECKER)
48      do
49        -- TODO: Add agents for the features in section `Rule
              checking' here.
50      end
51
52  feature {NONE} -- Rule checking
53
54    -- TODO: Add the AST processing here.
55
56  feature -- Properties
57
58    title: STRING_32
59      do
60          -- TODO: Add the title of your rule here.
61        Result := "(Your title)"
62      end
63
64      -- TODO: Add the ID of your rule here. Should be unique!
65    id: STRING_32 = "(YourID)"
66
67    description: STRING_32
```

```
68      do
69          -- TODO: Add the rule description here.
70        Result := "(Your description)"
71      end
72
73    format_violation_description (a_violation: attached
          CA_RULE_VIOLATION; a_formatter: attached TEXT_FORMATTER)
74      do
75        -- TODO: Add a formatted description of a concrete violation
              of this rule here.
76      end
77
78  end
```

We now have a closer look at the various parts of a rule class.

Initialization

Calling `make_with_defaults` initializes the attributes to their default values and makes sure that the class invariant is true. If you want to set an attribute to a custom value you can do so by setting it after the call to `make_with_defaults`.

The creation procedure from the template takes an argument of type `PREFERENCE_MANAGER`. This is used for initializing preferences that are specific to your rule. Such preferences usually represent integral or boolean values. If you do *not* need any custom preferences then you can leave out the argument `a_pref_manager` of `make` and you can remove the whole `initialize_options` feature.

AST Processing

The main part of your rule implementation consists of checking the source code for rule violations. Say, for example, that you want to check `if` instructions to have certain properties. Then you would add a feature like `process_if (a_if_ast: IF_AS)` to the section *Rule checking*. Also, you would need to modify the `register_actions` feature by adding the line

```
a_checker.add_if_pre_action (agent process_if).
```

Of course you may register as many such agents as you want.

Properties

The *title* and the *description* of the rule may be constant strings, they may also be localized strings. The *rule ID* must be unique among all rules. It should not contain spaces and should be reasonably short. The main rules that come with Inspector Eiffel have IDs that are numbered from *CA001* to *CA999* (many of which are not used).

Listing 4.20: Using TEXT_FORMATTER.

```
1 a_formatter.add ("Feature ")
2 if attached {STRING_32} a_violation.long_description_info.first as
      l_feat_name then
3   a_formatter.add_feature_name (l_feat_name, a_violation.
        affected_class)
4 end
5 a_formatter.add (" is very long.")
```

Formatted Violation Description

A rule should be able to produce a formatted description of a concrete rule violation. This description is for example used in the Inspector Eiffel tool panel of the GUI. There, class names and feature names are enabled for pick-and-drop. Variable names, numbers, and strings will be displayed in a nice way, too. In addition, this description is used in command line mode. In order to produce normal, unformatted text, use {TEXT_FORMATTER}.add. For adding formatted elements use features like:

- {TEXT_FORMATTER}.add_local,
- {TEXT_FORMATTER}.add_feature_name,
- and similar.

You should store all the data you need for this description (variables names, numbers, etc.) in {CA_RULE_VIOLATION}.long_description_info. format_violation_description can then retrieve this data for the formatted output. Listing 4.20 shows a simple example of producing a formatted description.

4.4.2 More Customized Rules

For rules that do not fit into a simple AST visitor scheme you best inherit your rule from CA_STANDARD_RULE, too. You can for example register agents that are called when a *class* or a *feature* is processed. Based on these agents you can perform your customized analysis on the classes and/or features. Using *multiple inheritance* or *aggregation* it should hardly be a problem to include any functionality you need for your analysis.

4.4.3 Accessing Type Information

The AST classes do not contain *type information*. Suppose your rule processes function calls. Feature calls in the AST do not contain any information on the types, such as the type of the result.

The Inspector Eiffel framework however provides functionality to retrieve the type of AST nodes. Before the analyzer lets a class be analyzed by

all the rules it computes the types of the AST nodes of a class. Hence this data will be available to your rule afterwards.

While your rule is being checked you can retrieve the type of node `a_node` from feature `a_feature` by calling

```
current_context.node_type (a_node: AST_EIFFEL a_feature: FEATURE_I).
```

`{CA_RULE}.current_context` is of type `CA_ANALYSIS_CONTEXT` and contains other information about current rule checking, too, such as the currently processed class or the matchlist for this class.

4.4.4 Accessing the Control Flow Graph

Some kinds of static code analysis need and use the *control flow graph* of a program. The INSPECTOR EIFFEL framework supports rules that use the control flow graph. If there is at least one such rule, INSPECTOR EIFFEL computes the control flow graph of the procedures of the analyzed class before letting the *rule* check this class.

Worklist Algorithms

Control flow graph rules iterate over the control flow graph. They do it using a *worklist*—a list of CFG edges that remain to be processed. At the beginning, the worklist contains all edges of the control flow graph. The algorithm will pick edges from the worklist for processing in an arbitrary order. The iteration stops as soon as there are no more edges left in the worklist. How will the worklist get smaller? Each edge that is processed is removed from the worklist. After processing you will have to decide dynamically whether to add all the outgoing (or incoming, depending on the direction) edges to the worklist. Like this you can take the fact into account that some analyses need certain edges to be processed more than once (a fixpoint iteration is such an example).

Implementation

A control flow analysis may iterate in either direction. For a forward-directed analysis inherit your rule from `CA_CFG_FORWARD_RULE`, for a backward analysis use `CA_CFG_BACKWARD_RULE` instead. In either case you will then have to implement the following deferred features:

`initialize_processing (a_cfg: attached CA_CONTROL_FLOW_GRAPH)` — This is called before a routine is processed using the worklist. Essentially you may use it to initialize and prepare all the data structures you will need during analysis.

`visit_edge (a_from, a_to: attached CA_CFG_BASIC_BLOCK): BOOLEAN` — This will be called when an edge is being visited. Here, you can put the analysis. If you let `Result = False` then no further edges will be added to the worklist. If in contrary you let `Result = True` then edges will be added to the worklist: In a *forward* analysis all the *outgoing* edges of the current one will be added; in a *backward* analysis all the *incoming* edges will be added.

Non-Worklist Algorithms

If your control flow graph does not fit into the structure of an algorithm as described above you may directly inherit from `CA_CFG_RULE` and implement the feature `process_cfg (a_cfg: attached CA_CONTROL_FLOW_GRAPH)` (in addition to the features explained above). In this case you do not have to use a worklist; basically you can process the control flow graph in any way you want.

4.5 UI Implementation

4.5.1 Graphical User Interface

The classes of the graphical user interface of Inspector Eiffel are all located in the `interface` cluster of EiffelStudio, in the subfolder `graphical` → `tools` → `code_analysis`. The following is a short overview of what the single classes do:

`ES_CODE_ANALYSIS_TOOL` — Represents the Inspector Eiffel GUI tool. The relevant things it contains are the tool title and the icon.

`ES_CODE_ANALYSIS_TOOL_PANEL` — The graphical panel for the Inspector Eiffel tool. It contains buttons, labels, the rule violations table view, and other user interface elements.

`ES_CODE_ANALYSIS_COMMAND` — The command to launch Inspector Eiffel. It can be added to toolbars and menus. It can be executed using stones. This class also handles the caching (see Section 4.5.1).

`ES_CODE_ANALYSIS_BENCH_HELPER` — A helper class for the integration of Inspector Eiffel. It contains shared instances of `CA_CODE_ANALYZER` and `ES_CODE_ANALYSIS_COMMAND`, which are used by the GUI.

`ES_CA_SHOW_PREFERENCES_COMMAND` — The command is used by the *Preferences* button in the panel.

`ES_CA_FIX_EXECUTOR` — This class fixes a rule violation that has been found by Inspector Eiffel.

Figure 4.2 shows the relevant class relations for the Inspector Eiffel GUI.

Caching

It is a common case that GUI users run Inspector Eiffel again after having made some changes to the code. We do not need to analyze the same unchanged code again and again. Therefore Inspector Eiffel caches the results in memory. This only applies to the GUI mode.

Inspector Eiffel uses cached results exactly in one case: when the whole system is analyzed and the previous analysis was on the whole system, too.

The caching functionality is implemented in `ES_CODE_ANALYSIS_COMMAND`. When the command for analyzing the system is executed, the timestamps of

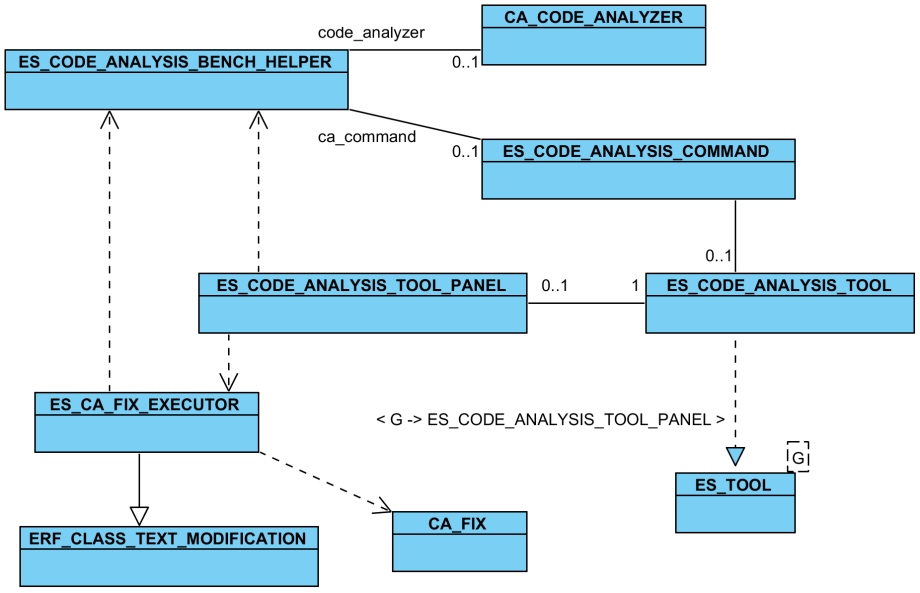

Figure 4.2: The relevant class relations of the Inspector Eiffel GUI.

the last modification of the classes are stored in

```
analysis_timestamp : HASH_TABLE [INTEGER, CLASS_I]
```

before the analysis. Note that the cached results (the rule violations) themselves are managed by `CA_CODE_ANALYZER`. The only difference to a non-cached analysis is that the rule violations are not deleted by `ES_CODE_ANALYSIS_COMMAND` before the next analysis. Then, in case the next command is also for analyzing the whole system, the current timestamps are compared to the stored timestamps. Any class that has been changed in the meantime will be analyzed again; for any unchanged class the rule violations are taken from the cache.

Example Command: Analyzing One Class

We will now roughly go through the code that is executed on the GUI part when the user wants to analyze a single class. As mentioned in Section 2.5.1, this can be done using the class context menu or by dropping the class stone on the button *Analyze Item*.

In either case `{ES_CODE_ANALYSIS_COMMAND}.execute_with_stone` is called, which delegates to `execute_with_stone_content` (shown in Listing 4.21). If there are modified unsaved windows, a save confirmation dialog is displayed. Eventually program flow passes on to `compile_and_analyze`, shown in Listing 4.22.

`eiffel_project.quick_melt` starts the compilation. A successful compilation is required for code analysis; otherwise nothing is analyzed. After compilation has succeeded we check if Inspector Eiffel is already running. If this is the case then a dialog is displayed. If on the other hand this last possible obstacle is not present we finally start analyzing by calling `perform_analysis`, shown in Listing 4.23 (the code that deals with stones other than classes is omitted).

Listing 4.21: Code for the command for analyzing a single class.

```
execute_with_stone (a_stone: STONE)
    -- Execute with 'a_stone'.
  do
    execute_with_stone_content (a_stone, Void)
  end

execute_with_stone_content (a_stone: STONE; a_content: SD_CONTENT)
    -- Execute with 'a_stone'.
  local
    l_save_confirm: ES_DISCARDABLE_COMPILE_SAVE_FILES_PROMPT
    l_classes: DS_ARRAYED_LIST [CLASS_I]
  do
      -- Show the tool right from the start.
    show_ca_tool

    if not eiffel_project.is_compiling then
      if window_manager.has_modified_windows then
        create l_classes.make_default
        window_manager.all_modified_classes.do_all (agent l_classes
            .force_last)
        create l_save_confirm.make (l_classes)
        l_save_confirm.set_button_action (l_save_confirm.
            dialog_buttons.yes_button, agent
            save_compile_and_analyze (a_stone))
        l_save_confirm.set_button_action (l_save_confirm.
            dialog_buttons.no_button, agent compile_and_analyze (
            a_stone))
        l_save_confirm.show_on_active_window
      else
        compile_and_analyze (a_stone)
      end
    end
  end
```

Listing 4.22: {ES_CODE_ANALYSIS_COMMAND}.compile_and_analyze

```
compile_and_analyze (a_stone: STONE)
    -- Compile project and perform analysis of stone 'a_stone'.
  local
    l_helper: ES_CODE_ANALYSIS_BENCH_HELPER
    l_dialog: ES_INFORMATION_PROMPT
  do
      -- Compile the project and only analyze if the compilation
          was successful.
    eiffel_project.quick_melt (True, True, True)
    if eiffel_project.successful then
      create l_helper
      if l_helper.code_analyzer.is_running then
        create l_dialog.make_standard (ca_messages.
            already_running_long)
        l_dialog.show_on_active_window
      else
        perform_analysis (a_stone)
      end
    end
  end
```

Listing 4.23: {ES_CODE_ANALYSIS_COMMAND}.perform_analysis

```
1  perform_analysis (a_stone: STONE)
2      -- Analyze 'a_stone' only.
3    local
4      l_helper: ES_CODE_ANALYSIS_BENCH_HELPER
5      l_scope_label: EV_LABEL
6    do
7        -- For simplicity let us assume that 'a_stone' does not
8        -- correspond to the system or is equivalent to it.
9      last_was_analyze_all := False
10
11     create l_helper
12     code_analyzer := l_helper.code_analyzer
13     code_analyzer.clear_classes_to_analyze
14     code_analyzer.rule_violations.wipe_out
15
16     l_scope_label := ca_tool.panel.scope_label
17
18     if attached {CLASSC_STONE} a_stone as s then
19       code_analyzer.add_class (s.class_i.config_class)
20       l_scope_label.set_text (s.class_name)
21       l_scope_label.set_foreground_color (create {EV_COLOR}.
             make_with_8_bit_rgb (140, 140, 255))
22       l_scope_label.set_pebble (s)
23       l_scope_label.set_pick_and_drop_mode
24       l_scope_label.set_tooltip (ca_messages.class_scope_tooltip)
25     elseif [...]
26       [...]
27     end
28
29     disable_tool_button
30     window_manager.display_message (ca_messages.status_bar_running)
31     code_analyzer.add_completed_action (agent analysis_completed)
32     code_analyzer.analyze
33   end
```

Listing 4.24: Invocation of the command-line version of code analysis.

```
1 elseif option.is_equal ("-code-analysis") then
2   l_at_args := arguments_in_range (current_option + 1,
        argument_count)
3   current_option := argument_count + 1
4   create {EWB_CODE_ANALYSIS} command.make_with_arguments (l_at_args
        )
```

At the start of the routine the code analyzer instance is retrieved from the helper class. All classes that may have been added before, are removed. All previous rule violations are removed as well. The `if` clause creates a stone for the *Last Scope* label in the graphical panel. Then, the button in the tool is disabled so that starting another analysis is prevented until the current one has completed. Finally, the analysis is started. As soon as the analysis has completed `{ES_CODE_ANALYSIS_COMMAND}.analysis_completed` is called. In this procedure the rule violations (and possibly the exceptions) are retrieved from the code analyzer and displayed in the list in the tool panel.

4.5.2 Command-Line Interface

The whole command-line functionality of Inspector Eiffel is located in the class `EWB_CODE_ANALYSIS`. It is located in the `tty` cluster of EiffelStudio. `EWB_CODE_ANALYSIS` is invoked by `ES`, the root class for the batch (command-line) version of EiffelStudio. In `ES`, the invocation looks as shown in Listing 4.24.

Any command-line arguments after `-code-analysis` are passed on to `EWB_CODE_ANALYSIS`. This class, in its creation procedure, processes the arguments as described in Section 2.4. Classes that were passed as command-line arguments are added to the analyzer. Then the actual execution happens in the procedure `execute`. `EWB_CODE_ANALYSIS` of course uses the `code_analysis` library and the previously described interface of `CA_CODE_ANALYZER`. After analysis a list of rule violations is output to the command-line. The relevant code is shown in Listing 4.25.

Listing 4.25: Command-line output.

```
across l_code_analyzer.rule_violations as l_vlist loop
  if not l_vlist.item.is_empty then
    l_has_violations := True
      -- Always sort the rule violations by the class they are
          referring to.
    output_window.add (ca_messages.cmd_class + l_vlist.key.name + "
        ':%N")

      -- See '{CA_RULE_VIOLATION}.is_less' for information on the
          sorting.
    across l_vlist.item as ic loop
      l_rule_name := ic.item.rule.title
      l_rule_id := ic.item.rule.id
      if attached ic.item.location as l_loc then
        l_line := ic.item.location.line.out
        l_col := ic.item.location.column.out
        output_window.add ("  (" + l_line + ":" + l_col + "): "
          + l_rule_name + " (" + l_rule_id + "): ")
      else -- No location attached. Print without location.
        output_window.add ("  " + l_rule_name + " (" + l_rule_id +
            "): ")
      end
      ic.item.format_violation_description (output_window)
      output_window.add ("%N")
    end
  end
end

if not l_has_violations then output_window.add (ca_messages.
    no_issues + "%N") end
```

Chapter 5

Conclusions

5.1 Conclusions

Our code analysis framework is capable of detecting many different issues in Eiffel source code. The fact that many implementations of rules are very short strongly supports our claim that Inspector Eiffel is simple to extend. The framework supports the functionality required for code analysis, as it has been used successfully by Inspector Eiffel. Inspector Eiffel has been successfully integrated in EiffelStudio and will be part of the next official release.

Our case studies show that our tool points to a significant number of actual code issues, and we were able to propose improvements. According to our statistics, *EiffelBase* has the highest code quality of all three external projects we investigated. This is a satisfactory result given that *EiffelBase* contains fundamental structures and algorithms of Eiffel. *EiffelVision* has only a slightly higher number of issues, whereas for the cluster from *EiffelStudio* we measured a significantly lower code quality.

5.2 Future Work

Amongst the rules in Appendix A there are 54 rules that we defined but did not implement. Adding these rules to Inspector Eiffel will make it more useful and more complete. Some of the drafted rules are already in the process of being implemented by other people.

Enabling automatic fixes for more rules—also for the ones that have already been implemented—is another important improvement. It leads to a greater assistance during the software development process. Additionally, we can imagine a more interactive fixing implementation: One could for example see a preview of the code as it will look once it has been fixed; one should be able to undo fixes and make the rule violation appear again.

We could make the connection to the code editor closer. Rule violations

could be marked directly in the code, making the code refinement process more visual and more interactive (similar functionality can be seen for example in [21]).

We think that a final goal of assistance and verification tools is to be integrated well enough so that they seem unseparated. If they do not appear as a separate tool but as an integral part of the IDE (i. e., the editor, or whatever it will be in the future of programming) then the hurdle and fear of constantly using the tools while programming might disappear.

5.3 Related Work

EiffelStudio has other components that cover analysis and testing of software: *AutoProof*, *AutoTest* and *AutoFix*. *AutoProof* statically verifies programs using Hoare-style proofs [16]. *AutoTest* is a dynamic analysis tool that does random testing [9]. *AutoFix* [17] automatically generates and validates fixes for software faults based on the results of *AutoTest*.

Several tools use static program analysis to detect code issues in languages other than Eiffel. They support Java, C# and VB.NET, as well as other languages. *SonarCube*^{TM} [20] is an open platform to manage code quality, thereby it uses static program analysis extensively. It supports Java, C#, C++, and many more languages. *PMD* [19] is a rule-based static code analyzer for Java, JavaScript, XML, and XSL. *FxCop* [22] from Microsoft performs static analysis of .NET code. It analyzes the object code rather than the source code. It was published both as a standalone tool and as a part in some integrated development environments from Microsoft. *ReSharper* from the company *JetBrains* [21] is a productivity tool for *Microsoft Visual Studio* that contains rule-based static analysis as one of its main features.

Appendix A

Rules

The following rules were proposed during the project, mostly before the implementation of Inspector Eiffel started. Hoping that the list is as comprehensive as possible—related to the Eiffel language—, it is not (and probably will never be) complete. One may always come up with more kinds of static analysis that would fit in our framework. Still we think that the important issues are all covered. Some rules might be controversial, especially the ones which are marked to be disabled by default and the ones that concern programming style. We are fully aware that programming style is to some extent a matter of taste.

The rules that have been implemented have the number in bold text[1].

Although many of the rules are specific to the Eiffel language and method, this list was partially inspired by our experience with the Java and C# static code analysis tools [19] and [21].

Due to the many properties a rule has, there are *two* tables which contain the exact *same rules*. The first table contains descriptions and classifications, whereas the second table shows code examples and lists the options (parameters) of the rules.

[1] The order of the rules is arbitrary. The numbering evolved during the project. Missing numbers are intentional; some numbers are missing due to nonambiguous numbering during the project.

A.1 List of Rules with Description and Classification

Abbreviations:

En. – Enabled Is the rule enabled by default?

Sev. – Severity The proposed rule severity; (W)arning, (S)uggestion, or (H)int. (We did not classify any rule as an *error*.)

Appl. – Applicability Non-library classes (NL) and/or library classes (L).

#	Rule	Scope	Description	En.	Sev.	Appl.
1	Self-assignment	instruction	Assigning a variable to itself is a meaningless instruction due to a typing error. Most probably, one of the two variable names was misspelled. One example among many others: the programmer wanted to assign a local variable to a class attribute and used one of the variable names twice. *Cf. rule #71.*	yes	W	NL, L
2	Unused argument	feature	A feature should only have arguments which are actually needed and used in the computation.	no	W	NL, L
3	Feature never called	system	There is no use for a feature that is never called by any class (including the one where it is defined). — *For attributes, see also rule #19.* — *A fix is implemented, which removes the corresponding feature.*	no	W	NL
4	No command-query separation (possible function side effect)	class	To the client of a class it is not important whether a query is implemented as an attribute or as a function. When a class evolves an attribute may be changed into a function, for example. A function should never change the state of an object. A function containing a procedure call, assigning to an attribute, or creating an attribute is a strong indication that this principle is violated. This rule applies exactly in these three last-mentioned cases. There are rather exceptional but sometimes useful class designs in which the externally visible state of an object (i. e. the values of exported queries) does not change even though the function contains a rule-violating instruction. — *Basically ensuring that no query changes the state of the current object or calls a command.* — [11, p. 748ff.]	yes	W	NL, L

#	Rule	Scope	Description	En.	Sev.	Appl.
5	Object test with object test local on read-only variable (locals, object test locals, arguments)	feature	The `attached` syntax prevents the program from calling a feature on a void reference. This is useful for attributes for they might be set to void by another feature, either in the code before the critical call or even by another thread. For local variables and feature arguments it is unnecessary to let the attached keyword create a new and safe local reference. Instead one should use a Certified Attachment Pattern [10]. *A fix is implemented, which removes the object test local.*	yes	S	NL, L
6	Object test typing not needed	system	An object test is redundant if the static type of the tested variable is the same as the type (in curly braces) that the variable is tested for. *Cf. rule #5.*	yes	S	NL, L
7	Object test always failing	system	An object test will always fail if the type that the variable is tested for does not conform to any type that conforms to the static type of the tested variable. The whole `if` block will therefore never be executed and it is redundant.			
8	Non-reinitializing of variable used in a loop	feature	Forgetting to re-initialize a variable in a loop may lead to errors that are often difficult to detect (without code analysis). *If the variable `i` was used previously in the code, we most likely need to reinitialize it to a sensible value especially if the loop is part of a larger loop.*			
9	Useless contract with void-safety	class	If a certain variable is declared as attached, either explicitly or by the project setting "Are types attached by default?" then a contract declaring this variable not to be void is useless. This rule only applies if void safety is enabled in the project settings. *If code is compiled in void-safe mode, we should remove contracts related to checking void-safe properties.*			
10	High complexity of nested branches and loops	feature	With the number of nested branches or loops increasing, the code get less readable. If the branch and loop complexity is high then the code should be refactored in such a way as to reduce its complexity.	yes	W	NL, L

#	Rule	Scope	Description	En.	Sev.	Appl.
11	Many feature arguments	feature	A feature that has many arguments should be avoided since it makes the class interface complicated and it is not easy to use. The feature arguments may include options, which should be considered to be moved to separate features. Interfaces of features with a large number of arguments are complicated, in the sense for example that they are hard to remember for the programmer. Often many arguments are of the same type (such as `INTEGER`). So, in a call, the passed arguments are likely to get mixed up, too, without the compiler detecting it. Arguments where in most of the cases the same value is passed—the default value—are called options. As opposed to operands, which are necessary in each feature call, each option should be moved to a separate feature. The features for options can then be called before the operational feature call in order to set (or unset) certain options. If a feature for an option is not called then the class assumes the default value for this option. [11, p. 764ff.]	yes	W	NL, L
12	Missing creation procedure with no arguments	class	In most of the cases a class that has at least one creation procedure with arguments should also have a creation procedure with no arguments. Arguments of creation procedures most often are some initializing data or options. Normally, one should be able to create a class with empty initializing data, where the client can set or add the data later. For options, an argumentless creation procedure may assume default values.		S	
13	Creation procedure is exported and may be called after object creation	class	If the creation procedure is exported then it may still be called by clients after the object has been created. Usually, this is not intended and ought to be changed. A client might, for example, by accident call `x.make` instead of `create x.make`, causing the class invariant or postconditions of make to not hold any more.	yes	W	NL, L
14	Unused local variable	feature	A routine should only define local variables that are actually needed and used in the computation. *(Already done by the compiler.)*		W	
15	Double negation	instruction	A double negation appearing in an expression can be safely removed. It is also possible that the developer has intended to put a single negation and the instruction is erroneous.			
16	Empty loop	instruction	A loop with an empty body should be removed. In most of the cases the loop never exits.			
17	Empty conditional instruction	instruction	An empty conditional instruction is useless and should be removed.	yes	W	NL, L

#	Rule	Scope	Description	En.	Sev.	Appl.
19	Attribute never gets assigned	class	An attribute that never gets assigned will always keep its default value. *See also rule #3.*			
20	Variable not read after assignment	feature	An assignment to a local variable has no effect at all if the variable is not read after the assignment, and before it is reassigned or out of scope.	yes	W	NL, L
21	Loop invariant computation within loop	instruction	A loop invariant computation that lies within a loop should be moved outside the loop.			
22	Unreachable code	feature	Code that will never be executed should be removed. It may be there for debug purposes or due to a programmer's mistake. One example is a series of instructions (in the same block) which follows an assertion that always evaluates to false.			
23	Unneeded parentheses	instruction	Parentheses that are not needed should be removed. This helps enforcing a consistent coding style. *One might consider allowing expressions like in* `z := (z + 3)` *and only forbidding* `z := ((z + 3))` *and alike.*	yes	S	NL, L
24	`From-until` loop on `ITERABLE` can be reduced to `across` loop	instruction	A `from-until` loop iterating through an `ITERABLE` data structure from beginning to end can be transformed into a (more recommendable) `across` loop.	yes	S	NL, L
25	Semicolon to separate arguments	feature	Routine arguments should be separated with semicolons. Although this is optional, it is bad style not to put semicolons.	yes	S	NL, L
28	Two `if` instructions can be combined using short-circuit operator	instruction	Two nested `if` instructions, both not having an `else` clause, should be combined into a single `if` instruction using the short circuit `and then` operator.	yes	S	NL, L
29	Object test or non-void test always succeeds	instruction	For an attached variable object tests and non-void tests always succeed. The tests should be removed.			
30	Unnecessary sign operand	instruction	All unary operands for numbers are unnecessary, except for a single minus sign. They should be removed or the instruction should be checked for errors.			
31	Explicit inheritance from `ANY`	class	Inheritance with no adaptations from the `ANY` class need not explicitly be defined. This should be removed.		H	
32	Very long routine implementation	feature	A routine implementation that contains many instructions should be shortened. It might contain copy-and-pasted code, or computations that are not part of what the feature should do, or computation that can be moved to separate routines.	yes	W	NL, L

#	Rule	Scope	Description	En.	Sev.	Appl.
33	Very big class	class	A class declaration that is very large (not including inherited features) may be problematic. The class might provide features it is not responsible for.	yes	W	NL, L
34	High NPATH complexity	feature	NPATH is the number of acyclic execution paths through a routine. A routine's NPATH complexity should not be too high. In order to reduce the NPATH complexity one can move some functionality to separate routines. — [14] — *Cf. rule #91.*	yes	W	NL, L
35	Feature section not commented	class	A feature section should have a comment. This comment serves as caption and is used for example by the 'Features' panel.	no	S	NL, L
36	Feature not commented	feature	A feature should have a comment. Feature comments are particularly helpful for writing clients of this class. To the programmer, feature comments will appear as tooltip documentation.	no	S	NL, L
37	Undesirable comment content	comment	Under some circumstances it might be desirable to keep a certain language level. Imaginable cases include source code that will be visible to people outside the company or that will even be released publicly.	no	W	
38	Empty argumentless creation procedure can be removed	class	If there exists only one creation procedure in a class and this procedure takes no arguments and has an empty body then the creation procedure should be considered to be removed. Note that in this case all the clients of the class need to call `create c` instead of `create c.make`, where `c` is an object of the relevant class and `make` is its creation procedure.	no	S	
39	Indication of high class coupling	class	A too high number of different types appearing in a class (types of attributes, argument, local variables, results) is a strong indicator for high coupling between classes.		W	
40	Command call on object returned from query (high coupling)	feature	On an object that has been obtained from a previous query on another object no command should be called. This is one of the forms of high coupling between objects, which should be avoided. One way to avoid such chained calls is by creating delegate methods.	no	W	
41	Boolean result can be returned directly	feature	For a boolean result there is no need for an `if`/`else` clause with `Result := True` and `Result := False`, respectively. One can directly assign the `if` condition (or its negation) to the result.	yes	S	NL, L

#	Rule	Scope	Description	En.	Sev.	Appl.
42	Unneeded comparison of boolean variables or queries	instruction	In expressions, boolean variables or queries need not be compared to `True` or `False`. *There may be other cases where a boolean expression can be simplified in a reasonable way.*	yes	S	NL, L
43	Deeply nested `if` instructions	instruction	Deeply nested `if` instructions make the code less readable. They should be avoided; one can refactor the affected code by changing the decision logic or by introducing separate routines.	yes	W	NL, L
44	Many instructions in an `inspect` case	instruction	A case of an `inspect` construct containing many instructions decreases code readability. The number of instructions should be lowered, for example by moving functionality to separate features.	yes	S	NL, L
45	Comparison of `{REAL}.nan`	instruction	To check whether a `REAL` object is "NaN" (not a number) a comparison using the `=` symbol does not yield the intended result. Instead one must use the query `{REAL}.is_nan`.		W	
46	Avoid 'not equal' in `if-else` instructions	instruction	Having an `if-else` instruction with a condition that checks for inequality is not optimal for readability. Instead an equality comparison should be made. Refactoring by negating the condition and by switching the instructions solves this issue.	yes	S	NL, L
47	Void-check using `is_equal`	instruction	Checking a local variable or argument to be void should not be done by calling `is_equal` but by the `=` or `/=` operators.		W	
48	Attribute can be made constant	class	An attribute that is assigned the same value by every creation procedure but not assigned by any other routine can be made constant. *Cf. rule #19.*		S	
49	Comparison of object references	instruction	The `=` operator always compares two object references by checking whether they point to the same object in memory. In the majority of cases one wants to compare the object states, which can be done by the `~` operator.		W	
50	Local variable only used for result	feature	In a function, a local variable that is never read and that is not assigned to any variable but the result can be omitted. Instead the result can be directly used.			
51	Empty and uncommented routine	feature	A routine which does not contain any instructions and has no comment too indicates that some implementation might be missing there. *The routine may also be one of the creation procedures.*		H	

#	Rule	Scope	Description	En.	Sev.	Appl.
52	Number of elements of a structure is compared to zero	instruction	In a data structure, comparing the number of elements to zero can be transformed into the boolean query `is_empty`. *We must check for all descendants of `FINITE`, since both `count` and `is_empty` are defined there.*	yes	S	NL, L
53	Empty routine in deferred class	feature	A routine with an empty body in a deferred class should be considered to be declared as deferred. That way it will not be forgotten to implement the routine in the descendant classes and errors can be avoided.	no	S	NL, L
54	Attribute is only used inside a single routine	class	An attribute that is only used inside a single routine of the class where it is defined (and that is not read by any other class) can be transformed into a local variable.	yes	S	NL, L
55	Result of structure is void	feature	In some cases, a query that returns a data structure is not returning any elements. Then, instead of setting the Result to void one should consider to assign the Result an empty data structure instead.	no	S	
56	`inspect` instruction has few branches	instruction	In order to increase readability, an `inspect` instruction that contains only very few branches should be converted to an `if-elseif` instruction.		S	
57	Simplifiable boolean expression	instruction	Some negated boolean expressions can be simplified using the inverse comparison operator. — `not (a < b)` → `a >= b` — `not (a <= b)` → `a > b` — `not (a > b)` → `a <= b` — `not (a >= b)` → `a < b` — `not (a = b)` → `a /= b` — `not (a /= b)` → `a = b` — `not (a ~ b)` → `a /~ b` — `not (a /~ b)` → `a ~ b`	yes	S	NL, L
58	"God" class		A "God" class is a large and complex class that heavily accesses data of other simpler classes and has a low cohesion between its features. Such a class is potentially harmful to the design of the system. [7, p. 80ff.]			
59	Empty `rescue` clause	feature	An empty `rescue` clause should be avoided and leads to undesirable program behavior.		W	

#	Rule	Scope	Description	En.	Sev.	Appl.
60	`inspect` instruction has no `when` branch	instruction	An `inspect` instruction that has no `when` branch must be avoided. If there is an `else` branch then these instructions will always be executed: thus the multi-branch instruction is not needed. If there is no branch at all then an exception is always raised, for there is no matching branch for any value of the inspected variable.		W	
61	Very short identifier	instruction	A name of a feature, an argument, or a local variable that is very short is bad for code readability.	no	S	NL, L
62	Very long identifier	instruction	A name of a feature, an argument, or a local variable that is very long is bad for code readability.	no	S	NL, L
63	Class naming convention violated	instruction	Upholding naming conventions is one of the elements of a consistent coding style and enhances readability. *Since this convention is very widespread in the Eiffel world, providing options for alternative conventions (*CamelCase *etc.) is not intended.*	no		
64	Feature naming convention violated	instruction	Upholding naming conventions is one of the elements of a consistent coding style and enhances readability. *Since this convention is very widespread in the Eiffel world, providing options for alternative conventions (*CamelCase *etc.) is not intended.*			
65	Local variable naming convention violated	instruction	Upholding naming conventions is one of the elements of a consistent coding style and enhances readability.			
66	Argument naming convention violated	instruction	Upholding naming conventions is one of the elements of a consistent coding style and enhances readability.			
67	Formal generic parameter name has more than one character	class	Names of formal generic parameters in generic class declarations should only have one character.			
68	Object creation within loop	instruction	Creating objects within a loop may decrease performance. On such an occurrence it should be checked whether the object creation can be moved outside the loop.		H	
70	Creation procedure initializes attribute to default value	class	Assigning an attribute its default value (e. g. `0` for `INTEGER`) in creation procedures is not needed. Default values are assigned during object creation anyway. This instruction can be removed.		H	
71	Self-comparison	instruction	An expression comparing a variable to itself always evaluates to the same boolean value. The comparison is thus redundant. In an `until` expression it may lead to non-termination. Usually it is a typing error. *Cf. rule #1.*	yes	W	NL, L

#	Rule	Scope	Description	En.	Sev.	Appl.
72	Implicit string conversion	instruction	Concatenating a string variable (`READABLE_STRING_GENERAL`) and a string literal causes an implicit conversion of the variable into a `STRING_8` object. Through the conversion, unicode characters will get lost.		W	
73	Comment not well phrased	comment	The comment does not end with a period or question mark. This indicates that the comment is not well phrased. A comment should always consist of whole sentences. *Cf. rule #74.*			NL, L
74	Indexing clause not well phrased	class	Some class indexing clauses such as `description` should be well phrased, i. e. consist of whole sentences. *Cf. rule #73.*			L
75	Exported feature never called outside class	system	An exported feature that is used only in unqualified calls may be changed to secret.			NL
76	Incomplete feature comment	feature	The feature comment should mention all the feature arguments.			
77	Unused inheritance	system	A class has an inheritance link that is used neither for implementation nor for polymorphism. This inheritance link should be removed.			
78	Unused subtyping	system	A class uses an inheritance link only for implementation. Consider making this inheritance secret.			
79	Unneeded accessor function	class	In Eiffel, it is not necessary to use a secret attribute together with an exported accessor ('getter') function. Since it is not allowed to write to an attribute from outside a class, an exported attribute can be used instead and the accessor may be removed.			
80	TODO	instruction	A comment line starting with the string "TODO" or "To do" means remaining work to be done.	yes	S	NL, L
81	Feature can be moved to ancestor	system	If all direct descendants of a class have the same implementation of a feature then the feature can be moved to the common ancestor class.			NL, L
82	Missing `is_equal` redefinition	class	The class defines `{HASHABLE}.hash_code`, but does not redefine `is_equal`. `is_equal` may need to be redefined.	yes	W	NL, L
83	Unneeded expanded type creation	instruction	An object of an expanded type is explicitly created. Normally the creation is performed automatically by run-time.			

#	Rule	Scope	Description	En.	Sev.	Appl.
84	Class can be made expanded	class	An effective class without attributes that is used only as a collection of features may be declared as expanded to avoid explicit object creation to call these features.			
85	Unneeded helper variable	feature	A variable that is assigned a value only once and is then used only once can be replaced with the expression that computes this value. This applies as long as the line where the expression is inserted will not have too many characters. *Cf. rule #86.*	yes	S	NL, L
86	Introduce helper variable	feature	Common subexpressions may need to be refactored by assigning their value to a local variable to reduce code complexity and make the values of the expressions explicit. Also, whenever a line has too many characters, consider extracting the longest expression into a local helper variable. *Cf. rule #85.*		S	NL, L
87	Mergeable conditionals	feature	Successive conditional instructions with the same condition can be merged.	yes	S	NL, L
88	Mergeable feature clauses	class	Feature clauses with the same export status and comment maybe merged into one.			
89	Explicit redundant inheritance	class	Explicitly duplicated inheritance links are redundant if there is no renaming, redefining, or change of export status. One should be removed. *Cf. rule #90.*		S	NL, L
90	Implicit redundant inheritance	class	Implicitly duplicated inheritance links are redundant if there is no renaming, redefining, or change of export status. The direct ancestor can be removed. *Cf. rule #89.*		S	NL, L
91	High cyclomatic complexity	feature	Cyclomatic complexity measures the number of linearly independent paths through a routine. A routine's cyclomatic complexity should not be too high. In order to reduce the cyclomatic complexity one can move some functionality to separate routines. — [8] — *Cf. rule #34.*			
92	Wrong loop iteration	instruction	Often, `from-until` loops use an integer variable for iteration. Initialization, stop condition and the loop body follow a simple scheme. A loop following this scheme but violating it at some point is an indication for an error.	yes	W	NL, L

A.2 List of Rules with Sample Code

#	Rule	Sample Code	Replace By	Options [default]
1	Self-assignment	`a := a`	Replace LHS or RHS by intended variable.	

2	Unused argument	`feature double (n: INTEGER): INTEGER`
		`do`
		`  Result := 2 * 2`
		`end`
			Remove the argument (mind the callers) or fix the code.	

3	Feature never called			

4	No command-query separation (possible function side effect)	`feature width: INTEGER`
		`do`
		`  height := height + 10`
		`  Result := 100 - height`
		`end`
			Remove assignment to attribute `height`.	

5	Object test with object test local on read-only variable (locals, object test locals, arguments)	`if attached a_local as another_local then`	`if a_local /= Void then`
			or
			`if attached a_local then`	

6	Object test typing not needed	`if attached {FOO} foo as l_foo`		
7	Object test always failing	`if attached {PERSON} a_string as l_person then`	Remove whole `if` clause.	

8	Non-reinitializing of variable used in a loop	`from (...)  until (...)  loop`
		`(...)`
		`  i := i + 1`
		`end`		

9	Useless contract with void-safety	`feature foo (a: attached FOO)`
		`  require a /= Void`
		`(...)`
			Remove useless contract.	

#	Rule	Sample Code	Replace By	Options [default]

10	High complexity of nested branches and loops	`if a > 0 then`
		`  from j = 1 until j >= s loop`
		`  from k = 7 until k < 0 loop`
		`  if enable_check = True then`
		`  foo (k, j-1)`
		`  if log_level = 3 then`
		`  foobar`
		`  end`
		`  else`
		`  bar (2 * j)`
		`  end`
		`  k := k - 1`
		`  end`
		`  j := j + 1`
		`  end`
		`end`
				— Complexity threshold [5]

11	Many feature arguments	Client side [11, p. 766]:
		`my_document.print (printer_name, paper_size, color_or_not, postscript_level, print_resolution)`
			Client side [11, p. 767]:
			`my_document.set_printing_size (paper_size)`
			`my_document.set_color`
			`my_document.print`
				— # arguments threshold [4]

12	Missing creation procedure with no arguments			

#	Rule	Sample Code	Replace By	Options [default]

13	Creation procedure is exported and may be called after object creation	`create`
		`  make`
		`feature`
		`  make`
		`    do`
		`      (...)`
		`    end`
		`  other`
		`    do`
		`      (...)`
		`    end`
			`create`
			`  make`
			`feature {NONE} -- Init`
			`  make`
			`    do`
			`      (...)`
			`    end`
			`feature -- Other`
			`  other`
			`    do`
			`      (...)`
			`    end`	

14	Unused local variable			
15	Double negation	`if l_some_boolean and (not (not l_other)) then`	`if l_some_boolean and l_other then`	
16	Empty loop	`from j := 1 until j > 100 loop` `end`	Remove it.	
17	Empty conditional instruction	`if x And (a Or b) then` `end`	Remove it.	
18	Unused routine			
19	Attribute never gets assigned			

20	Variable not read after assignment	`l_x := 3`
		`l_a := x.get_width (True)`
		`l_x := x.get_height`
			Remove first line, for example.	

21	Loop invariant computation within loop	`across some_list as l_list loop`
		`  foo (l_list.item)`
		`  a := 4`
		`end`
			`across some_list as l_list loop`
			`  foo (l_list.item)`
			`end`
			`a := 4`	

#	Rule	Sample Code	Replace By	Options [default]

22	Unreachable code	`(...)`
		`check False end`
		`x := 5`
		`(...)`
			Remove code after `check False end`.	

23	Unneeded parentheses	`if (z > 3) then z := (z - 5) end`	`if z > 3 then z := z - 5 end`	

24	`from-until` loop on `ITERABLE` can be reduced to `across` loop	`from list.start`
		`until list.after`
		`loop`
		`  (...)`
		`  list.forth`
		`end`
			`across list as l_list`
			`loop`
			`  (...)`
			`end`	

25	Semicolon to separate arguments	`f (a: INTEGER b: STRING) do end`	`f (a: INTEGER; b: STRING) do end`	

28	Two `if` instructions can be combined using short-circuit operator	`if l_user /= Void then`
		`  if l_user.age >= 18 then`
		`  foo`
		`  end`
		`end`
			`if l_user /= Void and then l_user.age >=`
			`18 then`
			`  foo`
			`end`	

29	Object test or non-void test always succeeds	`if l_attached_var /= Void then`
		or
		`if attached attached_attribute as l_a then`
			Remove conditional.	

30	Unnecessary sign operand	`x := + 4`
		`y := - -7`
		`z := + - 9`
			`x := 4`
			`y := 7`
			`z := -9`	

31	Explicit inheritance from `ANY`	`class FOO inherit ANY`	`class FOO`	
32	Very long routine implementation			— # instructions threshold [70]

#	Rule	Sample Code	Replace By	Options [default]
33	Very big class			— # features threshold [20] — # instructions threshold [300]
34	High NPATH complexity			— NPATH complexity threshold [200]

35	Feature section not commented	`feature`
		`  set_size (size: INTEGER)`
		`  do`
		`  (...)`
		`  end`
		`  enable_transparency`
		`  do`
		`  (...)`
		`  end`
			`feature -- Set Appearance`
			`  set_size (size: INTEGER)`
			`  do`
			`  (...)`
			`  end`
			`  enable_transparency`
			`  do`
			`  (...)`
			`  end`	

36	Feature not commented	`feature enable_transparency`
		`  do`
		`  (...)`
		`  end`
			`feature enable_transparency`
			`  -- Enables transparency for the current element and all of its children.`
			`  do`
			`  (...)`
			`  end`	

37	Undesirable comment content	`  -- not working properly when running on xyz configuration (to be fixed)` `x.foo (bar)`		— List of problematic strings [“to be fixed”, ...] — Case sensitivity

38	Empty argumentless creation procedure can be removed	`create`
		`  make`
		`feature {NONE}`
		`  make`
		`  do`
		`  end`
			Remove `make` from create and feature sections.	

39	Indication of high class coupling			— # different types threshold [25]

#	Rule	Sample Code	Replace By	Options [default]

40	Command call on object returned from query (high coupling)	`l_bar = foo.bar`
		`bar.enable_transparency` or equivalently
		`foo.bar.enable_transparency`
			(Probably requires creation of delegate method.)
			`foo.enable_bar_transparency`	

41	Boolean result can be returned directly	`if x > 4 then`
		`  Result := False`
		`else`
		`  Result := True`
			`Result := (x <= 4)`	

42	Unneeded comparison of boolean variables or queries	`if l_some_boolean = False then`
		`  do_something`
		`end`
		`if a.has_children = True then`
		`(...)`
			`if not l_some_boolean then`
			`  do_something`
			`end`
			`if a.has_children then`
			`(...)`	

43	Deeply nested `if` instructions	`if attached {FOO} a.item as l_item then`
		`  if search_enabled then`
		`  if l_item.x > x_threshold then`
		`(...)`
				— Minimum depth [3]

44	Many instructions in an `inspect` case	`inspect l_days_left`
		`  (...)`
		`  when 3 then`
		`  [10 instructions]`
		`  (...)`
		`end`
			`inspect l_days_left`
			`  (...)`
			`  when 3 then`
			`  show_warning`
			`  (...)`
			`end`
			`(...)`
			`feature show_warning do`
			`  [10 instructions]`
			`end`
				— # instructions threshold [8]

45	Comparison of `{REAL}.nan`	`if my_real = {REAL}.nan then` `(...)`	`if my_real.is_nan then` `(...)`	

#	Rule	Sample Code	Replace By	Options [default]

46	Avoid 'not equal' in `if-else` instructions	`if x /= 3 then`
		`  (...)`
		`else`
		`  (...)`
		`end`
			`if x = 3 then`
			`  (...)`
			`else`
			`  (...)`
			`end`	

47	Void-check using `is_equal`	`if a_list.is_equal (Void) then` or `if not a_list.is_equal (Void) then`	`if a_list = Void then` or `if a_list /= Void then`	

48	Attribute can be made constant	`feature`
		`  make do some_constant := 42 end`
		`(...)`
		`  some_constant: INTEGER`
			`feature`
			`  make do end`
			`(...)`
			`  some_constant: INTEGER = 42`	

49	Comparison of object references	`if a = b then`	`if a.is_equal (b) then`	

50	Local variable only used for result	`create l_list.make`
		`if basic then`
		`  l_list.extend (x)`
		`else`
		`  l_list.extend (y)`
		`Result := l_list`
			`create Result.make`
			`if basic then`
			`  Result.extend (x)`
			`else`
			`  Result.extend (y)`	

51	Empty and uncommented routine	`feature foo`
		`  do`
		`  end`
			`feature foo`
			`  -- Empty body intentional - can be redefined by descendant if needed.`
			`  do`
			`  end`	

52	Number of elements of a structure is compared to zero	`if l_list.count /= 0 then` `(...)`	`if not l_list.is_empty then` `(...)`	

53	Empty routine in deferred class	`compute_mean`
		`  do`
		`  end`
			`compute_mean`
			`  deferred`
			`  end`	

#	Rule	Sample Code	Replace By	Options [default]
54	Attribute is only used inside a single routine			

55	Result of structure is void	`feature`
		`  violations_list: LINKED_LIST`
		`  do`
		`  if not_started then`
		`  Result := Void`
		`  else`
		`  (...)`
			`feature`
			`  violations_list: LINKED_LIST`
			`  do`
			`  if not_started then`
			`  create Result.make`
			`  else`
			`  (...)`	

56	`inspect` instruction has few branches	`inspect l_list.count`
		`  when 1 then foo (l_list)`
		`  when 2 then bar (l_list)`
		`end`
			`if l_list.count = 1 then`
			`  foo (l_list)`
			`elseif l_list.count = 2 then`
			`  bar (l_list)`
				— Minimum # branches required [3]

57	Simplifiable boolean expression	`if not (level > 3) then` `(...)`	`if level <= 3 then` `(...)`	
58	"God" class			
59	Empty `rescue` clause			

60	`inspect` instruction has no `when` branch	`inspect l_id`
		`  else`
		`  foo (l_id)`
		`  bar`
		`end`
			`foo (l_id)`
			`bar`	

61	Very short identifier			— Minimum feature name length [3] — Minimum argument name length [3] — Minimum local name length [3] — Count argument prefix '`a_`' [true] — Count local variable prefix '`l_`' [true]

#	Rule	Sample Code	Replace By	Options [default]
62	Very long identifier			— Maximum feature name length [30] — Maximum argument name length [20] — Maximum local name length [20]
63	Class naming convention violated			
64	Feature naming convention violated			
65	Local variable naming convention violated			— Naming schema [‘`l_*`’]
66	Argument naming convention violated			— Naming schema [‘`a_*`’]
67	Formal generic parameter name has more than one character	`class OPTIMIZER [CLNT]` `(...)`	`class OPTIMIZER [G]` `(...)`	— Allow letters other than ‘G’ for classes with only one formal generic parameter [true]

68	Object creation within loop	`from j := 1 to n_items loop`
		`  create l_optimizer`
		`  l_optimizer.check (item.at (j))`
		`end`
			`create l_optimizer`
			`from j := 1 to n_items loop`
			`  l_optimizer.check (item.at (j))`
			`end`	

69	`until` expression in loop always evaluates to `False`	`from j := 10 until j > j loop`
		`  (...)`
		`  j := j - 1`
		`end`		

70	Creation procedure initializes attribute to default value	`make`
		`  do`
		`  count := 0`
		`  (...)`
		`  end`
			Remove `count := 0`.	

#	Rule	Sample Code	Replace By	Options [default]

71	Self-comparison	`if a > a then`
		`  (...)`
		`end`		

72	Implicit string conversion	`(target: READABLE_STRING_GENERAL`
		`unicode_name: STRING_GENERAL)`
		`target := "name:  " + unicode_name`		

73	Comment not well phrased	`-- checks size of structure`	`-- Checks the size of the structure.`	
74	Indexing clause not well phrased			
75	Exported feature never called outside class			
76	Incomplete feature comment			
77	Unused inheritance			
78	Unused subtyping			

79	Unneeded accessor function	`feature {NONE} -- Implementation`
		`  internal_value: INTEGER`
		`feature -- Properties`
		`  value: INTEGER`
		`  do Result := internal_value end`
			`feature -- Properties`
			`  value: INTEGER`	

80	TODO			
81	Feature can be moved to ancestor			
82	Missing is_equal redefinition			

#	Rule	Sample Code	Replace By	Options [default]

83	Unneeded expanded type creation	`feature`
		`  foo`
		`  local`
		`  j: INTEGER`
		`  do`
		`  create j`
		`  j := 2`
		`  (...)`
		`  if bar then`
		`  create j`
		`  end`
		`  (...)`
		`  end`
			`feature`
			`  foo`
			`  local`
			`  j: INTEGER`
			`  do`
			`  j := 2`
			`  (...)`
			`  if bar then`
			`  j := 0`
			`  end`
			`  (...)`
			`  end`	

84	Class can be made expanded			
85	Unneeded helper variable	`l_name := "yoda"` `print (l_name)`	`print ("yoda")`	— Maximum allowed characters per line [80]

86	Introduce helper variable	`a_formatter.add_feature_name`
		`(a_violation.long_description_info.first,`
		`a_violation.affected_class)`
			`l_inf :=`
			`a_violation.long_description_info.first`
			`a_formatter.add_feature_name (l_inf,`
			`a_violation.affected_class)`
				— Maximum allowed characters per line [80]

87	Mergeable conditionals	`if a then`
		`  foo`
		`else`
		`  bar`
		`end`
		`if a then`
		`  baz`
		`end`
			`if a then`
			`  foo`
			`  baz`
			`else`
			`  bar`
			`end`	

#	Rule	Sample Code	Replace By	Options [default]

88	Mergeable feature clauses	`feature -- Properties`
		`  foo`
		`  do (...)  end`
		`(...)`
		`feature -- Properties`
		`  width: INTEGER`
			`feature -- Properties`
			`  foo`
			`  do (...)  end`
			`  width: INTEGER`
			`(...)`	

89	Explicit redundant inheritance	`class`
		`  APPLICATION`
		`inherit`
		`  ARGUMENTS`
		`  MY_CLASS`
		`  ARGUMENTS`
			`class`
			`  APPLICATION`
			`inherit`
			`  ARGUMENTS`
			`  MY_CLASS`	

90	Implicit redundant inheritance	`class`
		`  MY_CLASS`
		`inherit ARGUMENTS`
		`(...)`
		`class`
		`  APPLICATION`
		`inherit`
		`  ARGUMENTS`
		`  MY_CLASS`
			`class`
			`  MY_CLASS`
			`inherit ARGUMENTS`
			`(...)`
			`class`
			`  APPLICATION`
			`inherit MY_CLASS`	

91	High cyclomatic complexity			

92	Wrong loop iteration	`from j := 1 until j > 100 loop`
		`  foo (j)`
		`  j := j - 1`
		`end`
		or
		`from j := 1 until j < 100 loop`
		`  foo (j)`
		`  j := j + 1`
		`end`
			(both)
			`from j := 1 until j > 100 loop`
			`  foo (j)`
			`  j := j + 1`
			`end`	

List of Figures

Listings

Bibliography

[1] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2007.

[2] Dennis M. Breuker, Jan Derriks, and Jacob Brunekreef. Measuring static quality of student code. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 13–17, New York, NY, USA, 2011. ACM.

[3] Lamia Djoudi and William Jalby. Automatic analysis for managing and optimizing performance-code quality. In *Proceedings of the 2008 Workshop on Static Analysis*, SAW '08, pages 30–38, New York, NY, USA, 2008. ACM.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] Robert Green and Henry Ledgard. Coding guidelines: Finding the art in the science. *Commun. ACM*, 54(12):57–63, December 2011.

[6] Chaitanya Kothapalli, S. G. Ganesh, Himanshu K. Singh, D. V. Radhika, T. Rajaram, K. Ravikanth, Shrinath Gupta, and Kiron Rao. Continual monitoring of code quality. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 175–184, New York, NY, USA, 2011. ACM.

[7] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[8] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), December 1976.

[9] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Yi Wei, and E. Stapf. Programs that test themselves. *Computer*, 42(9):46–55, Sept 2009.

[10] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[11] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[12] Bertrand Meyer. More expressive loops for eiffel. `http://bertrandmeyer.com/2010/01/26/more-expressive-loops-for-eiffel/`, March 2014.

[13] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[14] Brian A. Nejmeh. Npath: A measure of execution path complexity and its applications. *Communications of the ACM*, 31(2), February 1988.

[15] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005.

[16] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Automatic verification of advanced object-oriented features: The autoproof approach. In *Tools for Practical Software Verification - LASER 2011, International Summer School*, volume 7682 of *LNCS*, pages 134–156. Springer, 2012.

[17] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72, New York, NY, USA, 2010. ACM.

[18] Eiffelbase in the eiffel documentation. `http://docs.eiffel.com/book/solutions/eiffelbase`, January 2014.

[19] Pmd. `http://pmd.sourceforge.net/`, March 2014.

[20] Sonarcube^{TM}. `http://www.sonarcube.org/`, March 2014.

[21] Jetbrains resharper. `http://www.jetbrains.com/resharper/`, March 2014.

[22] Microsoft fxcop. `http://www.microsoft.com/en-us/download/details.aspx?id=6544`, March 2014.