Chair of Software Engineering

Eiffel Inspector Improvements

Bachelor Thesis

Samuel Schmid
ETH Zurich
schmisam@student.ethz.ch

September 15, 2014 - March 15, 2015

Supervised by:
Julian Tschannen
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

Code quality is an important issue in software engineering. It is a topic which is hard to define because it is mostly a subjective matter. The Eiffel programming language encourages high code quality and also, with the addition of the Eiffel Inspector, provides means to detect code smells.

The Eiffel Inspector is a tool designed for static analysis of code. The framework is rule-based, meaning that every analysis of the code is based on a set of rules that define violations of code quality. With this tool's help developers can easily check their work for code quality issues and even apply automated fixes to violations of the rules offered by the tool. The automated fix mechanic changes the source code directly in the EiffelStudio IDE.

We extended the Eiffel Inspector by several new rules. All of the rules have been implemented in Eiffel and added to the EVE IDE, a research branch of the EiffelStudio IDE. We implemented user interface improvements like undoing the changes done by automated fixes and being able to select only a set of rules which are used in the analysis of the code. With this addition the Eiffel Inspector now contains 57 rules checking for code quality and adding new rules to the framework has been simplified with the introduction of the interface improvements.

Acknowledgments

First and foremost I would like to thank my supervisor Julian Tschannen for all his help and support during the thesis project.

Also I want to thank the Eiffel EVE Wiki page contributors for providing a tutorial for setting up EiffelStudio on MacOS. Special thanks also go to Emmanuel Stapf for his eager support with the setup on the Mac.

Last but not least I want to thank my parents for their moral and financial support during my time at ETH.

Samuel Schmid

Contents

Chapter 1

Introduction

1.1 The Eiffel Inspector

Good code quality is difficult to achieve and maintain, especially when working in a large team. It is one of the biggest challenges in software engineering [10, 11]. Every programmer has different views about what is *good code*. Having a coding style guideline document [9] to check against the code is a highly time-consuming task. So the only sensible solution that remains is to do automated testing [1, 4].

The EiffelStudio IDE provides the tool called the *Eiffel Inspector* [20] for automated testing of code quality. The Eiffel Inspector is a rule-based framework. For detecting code smells, it runs the code against a set of rules defining good code quality or coding style guidelines. After the analysis all the rule violations are presented to the developer with annotations as to where the violation was found in the code. There are fully automated tools which not only detect bad code but also provide automated fixing of the detected issues (without the user having to select any), but the Eiffel Inspector is only partially automated. The developer can choose which violations should be fixed automatically by the tool and which ones he wants to resolve himself.

This thesis is aiming at improving the Eiffel Inspector tool by adding new rules, adding automatic fixes for the rules where applicable, and improving the user interface. We also integrated the Eiffel Inspector with another tool in the Eiffel IDE, the Verification Assistant [17].

In Chapter 2 we discuss the newly added rules to the Eiffel Inspector. Each section contains a new rule and a short explanation of its implementation.

The additions to the user interface are shown in Chapter 3. We added two new buttons to improve the way the user can automatically fix violations in the code. Also we added a new tag for classes to make selective verification possible.

In Chapter 4 we discuss the integration of Eiffel Inspector with the Verification Assistant, a tool which provides a unified interface for various analysis and verifications tools (such as the Eiffel Inspector and other tools mentioned in Section 1.2).

Chapter 5 talks about the improvements made to the existing framework of the Eiffel Inspector. These include refactoring of rules, changing the folder hierarchy of the framework to be more sensible, and increasing the performance of the analysis.

Chapter 6 shows our conclusions and possible future work.

1.2 Related Work

The EiffelStudio IDE provides additional tools for automated testing and verification of software. AutoTest [12] is an analysis tool which uses random testing. AutoFix [19] then generates fixes for software faults found by AutoTest. AutoProof [18] verifies software using Hoare-style proofs with static program analysis. AutoProof and AutoFix are available in the EVE IDE.

Looking beyond the Eiffel programming language there are other tools for detecting bad code in languages like Java, C#, VB.NET, and many others as well. *SonarQube*^{TM} [16] is an open platform to manage code quality. It is a web-based application which covers over 20 different programming languages. *Pmd* [15] is a code analyser to find common programming flaws in Java, Javascript, XML, and XSL. *FxCop* [13] is a graphical user interface and command line tools for performing static code analysis of .NET code developed by Microsoft.

Chapter 2

New Rules

In this chapter we will present the new rules which we added to the Eiffel Inspector. Each section contains a new rule and talks about the implementation and the automated fix which can be applied to its violations.

There is an internal working document for the development of the Eiffel Inspector describing all of the rules [1]. Each section starts with its rule's description from that document.

2.1 Object test always failing

2.1.1 Rule

Description:

> *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.*

Listing 2.1: Example violation of object test always failing rule.

```
1 local
2   l_string: STRING
3 do
4   if attached {PERSON} l_string as l_person then
5       -- do something
6   end
7 end
```

An object test is a construct in Eiffel used to test the dynamic type of an object. In order for this test to be able to succeed there needs to be a common

[1] This document can be accessed upon request.

child between the type in the object test (here PERSON) and the static type of the variable (here STRING). Since this is most probably not true for the types PERSON and STRING we can see an example violation of this rule in Listing 2.1.

2.1.2 Implementation

This rule is implemented using the visitor pattern [8] over the abstract syntax tree created by the Eiffel compiler. Part of the implementation can be seen in Listing 2.2.

We visit all the OBJECT_TEST_AS nodes to check every object test in the program. In lines 2 and 3 we extract the types used in the test from the type recorder and run the feature has_common_child on them in line 5. This feature checks whether there is a common child for the type used in the object test and the static type of the variable being tested. If that is not the case, we create a violation of this rule for this class in line 6.

Listing 2.2: Part of the object test always failing rule implementation.

```
1 if attached a_ot.type then
2   l_type_1 := current_context.node_type (a_ot.expression)
3   l_type_2 := current_context.node_type (a_ot.type)
4
5   if not has_common_child (l_type_1, l_type_2) then
6     create_violation (a_ot)
7   end
8 end
```

2.1.3 Fix

Fixing violations of this rule is not as simple as stated in the rule description in Section 2.1.1. The condition of the if block might be connected with or keywords and therefore the condition might still hold, even though the object test itself always evaluates to False. There might not even be an if block, as the object test construct can be used anywhere where one can also use a BOOLEAN variable.

So we decided on the following procedure: If we have a violation of this rule, we simply replace the object test construct with the keyword False and leave the rest up to the Eiffel Inspector to find further code quality issues (e.g. an if block that reads if False then after applying this fix).

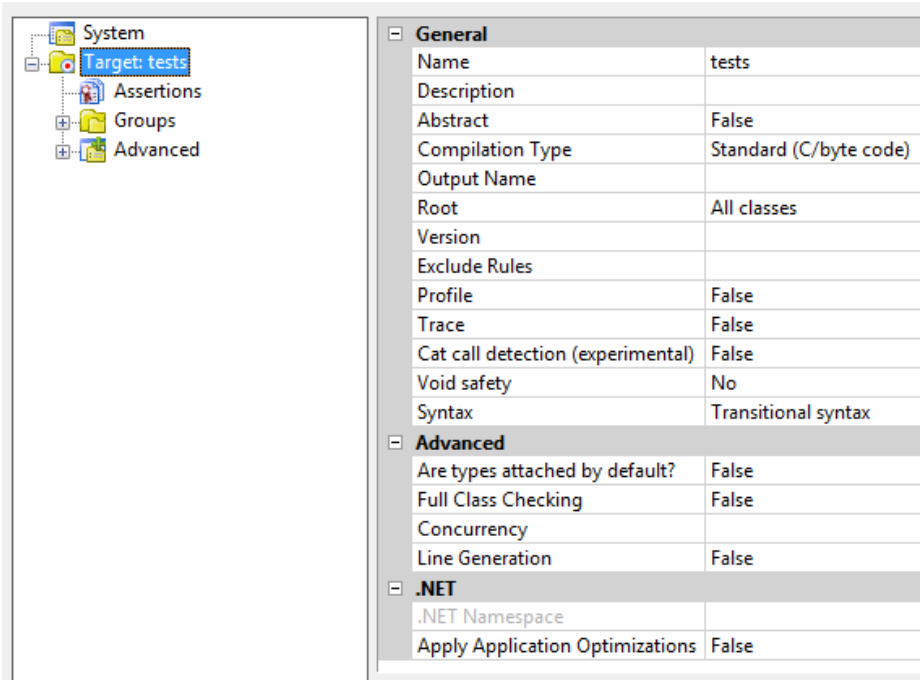

Figure 2.1: The project settings window of EiffelStudio.

2.2 Useless contract with Void-safety

2.2.1 Rule

Description:

> *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 the project setting for Void safety is set to "Complete".*

Every variable in Eiffel can be declared as an `attached` type. This means that the variable cannot be set to `Void` or set to anything that can be set to `Void`. There are two settings in the menu "Project Settings" in EiffelStudio. One of them is the *Void safety* setting, which we will look at later, and the other one is the *Are types attached by default?* setting as seen in Figure 2.1.

When the latter is set to *True*, then a normal type declaration of a variable like `l_string: STRING` would result in an `attached` type variable `l_string`, meaning that an assignment like `l_string := Void` would result in a compiler error. Now this does not mean that there are no more ways to create variables that are `Void` in fact, it is sometimes important to be able to assign `Void` to some variables.

That is why there is another keyword: `detachable`. It is basically the opposite of `attached`, meaning that any variable of a `detachable` type can be set to `Void` or set to anything that can be set to `Void`.

The other setting (Void safety) is a feature in Eiffel which ensures at compile time that your system will not attempt to apply a feature to a Void reference at run time. This is achieved using the aforementioned `attached` types.

This rule is only applied and checked if the class that is being checked has Void safety set to *Complete*, because the `attached` marks are only recorded when Void safety is available. Otherwise they are ignored because they are of no use to the compiler.

2.2.2 Implementation

Again we use the visitor pattern to find violations to this rule. We visit all the `FEATURE_AS` nodes to cover all the features in the class to be checked. There we iterate through possible pre- and or postconditions to see if any of them contain a `BIN_NE_AS` node. These nodes represent binary /= operations (not equal).

Now all that is left, is to check whether one side of the binary equation is `Void` and the other side a variable which has an `attached` type. If that is the case we create a violation for this rule.

2.2.3 Fix

Fixing violations of this rule is simple: We remove the violating pre- or postcondition and if it is the only contract in the block we remove the preceding keyword (`require` or `ensure`, depending on the condition) as well. Listings 2.3 and 2.4 show an example of an application of this automated fix.

Listing 2.3: Example violation of useless contract rule.

```
1 feature foo (a_1: attached A)
2   require
3     a_1 /= Void
4   do
5     a_1.cleanup
6   ensure
7     a_1.is_clean
8   end
```

Listing 2.4: Automated fix applied to violation in Listing 2.3.

```
feature foo (a_1: attached A)


  do
    a_1.cleanup
  ensure
    a_1.is_clean
  end
```

2.3 Object test or non-Void test always succeeds

2.3.1 Rule

Description:

> *For an attached variable object tests and non-Void tests always succeed. Also, objects tests that check if an entity is attached to a supertype always suceed. The tests should be removed.*

This rule is only applied and checked when the Void-safety setting of the project is set to *"Complete"* because it uses the `attached` marks for the variables to check for violations. See Section 2.2.1 for more information about `attached` and `detachable`.

When a variable has an `attached` type then both an object test without a type (which is basically only checking if the variable is `Void` or not) as well as an explicit non-Void test will aways succeed. These tests can either be removed or replaced with the boolean constant `True`.

2.3.2 Implementation

We use the visitor pattern to visit the correct nodes in the abstract syntax tree. We need to visit all the `OBJECT_TEST_AS` as well as `BIN_NE_AS` nodes to cover all the cases.

In the `OBJECT_TEST_AS` nodes we check if the variable being tested is of an `attached` type, and if that is the case we also need to make sure that the object test does not check for a dynamic type. If it is checking for a dynamic type, we cannot remove it because we cannot be sure that it will always succeed.

In the `BIN_NE_AS` nodes (which represent the binary /= operation) we look for an operation that has the keyword `Void` on one side, i.e. one that is testing if the other side is `Void`. Then we check if that other side is a variable of an `attached` type.

For each occurrence of a non-Void test or object test with a variable of an `attached` type we create a separate violation for this rule.

2.3.3 Fix

Since we do not know what the user is intending with the test and also because we could be in a deeply nested if-condition, the automated fix for this rule is simple: We replace the non-Void test or the object test construct with the keyword `True` and leave the rest up to the Eiffel Inspector. An example of the application of this fix can be seen in Listings 2.5 and 2.6.

Listing 2.7: Example violation of comparison of {REAL}.nan rule.

```
1 local
2   l_real: REAL
3 do
4   if l_real = {REAL}.nan then
5       -- do something
6   end
7 end
```

Listing 2.5: Example violation of object test always succeeds rule.

```
1 local
2   l_string: attached STRING
3 do
4   if l_string /= Void then
5     Result := l_string.length
6   end
7 end
```

Listing 2.6: Automated fix applied to violation in Listing 2.5.

```
local
  l_string: attached STRING
do
  if True then
    Result := l_string.length
  end
end
```

In this example we end up with an if block which is redundant because the condition always holds, and the next iteration of analysis in the Eiffel Inspector will create a violation for that.

2.4 Comparison of {REAL}.nan

2.4.1 Rule

Description:

> *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.*

In Listing 2.7 the user wants to check whether the local variable `l_real` is a real number or NaN (not a number). The class `REAL` provides the features `is_nan` and `nan` both of which can be accessed in a static way. The feature `is_nan` can be used to check whether a `REAL` variable is a NaN or not. The Feature `nan` on the other hand is used to represent a `REAL` variable which is NaN e.g. for assigning it to a `REAL` variable.

The problem is that the way NaN is implemented, every comparison with it will return `False`. Which means both `{REAL}.nan = {REAL}.nan` as well as `{REAL}.nan /= {REAL}.nan` are `False`. Therefore one should rather use the query `{REAL}.is_nan` on the variable to be tested.

Listing 2.8: Checking for real types.

```
if
  (l_type.is_equal ("REAL_32")
  or else l_type.is_equal ("REAL_64")
  or else l_type.is_equal ("REAL_32_REF")
  or else l_type.is_equal ("REAL_64_REF"))
then
```

2.4.2 Implementation

Violations of this rule are checked using the visitor pattern. We visit all the `BIN_EQ_AS` (binary comparison) nodes in the class in order to check all comparisons between two objects. We then check whether the binary equation contains a `REAL` type variable on one side and a `STATIC_ACCESS_AS` node which calls the feature `{REAL}.nan` on the other. This check needs to run twice because the user could either write

```
if l_real = {REAL}.nan
```

or use the boolean comparison the other way around and write

```
if {REAL}.nan = l_real
```

Checking for a `REAL` type variable turned out to be more complicated because Eiffel has four different classes for `REAL` variables. We ended up simply checking the name of the type as seen in Listing 2.8 which is sufficient since Eiffel class names are unique.

2.4.3 Fix

Fixing a violation of this rule is simple. One only needs to replace the violating binary comparison with the static call to `{REAL}.is_nan` on the same variable that has been used in the comparison. An example of a fix can be seen in Listings 2.9 and 2.10.

Listing 2.9: Example violation of {REAL}.nan comparison rule.

```
1 local
2   l_real: REAL
3 do
4   if l_real = {REAL}.nan then
5       -- do something
6   end
7 end
```

Listing 2.10: Automated fix applied to violation in Listing 2.9.

```
local
  l_real: REAL
do
  if l_real.is_nan then
      -- do something
  end
end
```

Listing 2.11: Example violation of local variable only used for Result rule.

```
1  local
2    l_list: LINKED_LIST
3  do
4    create l_list.make
5    if a_boolean then
6      l_list.extend (a_x)
7    else
8      l_list.extend (a_y)
9    end
10   Result := l_list
11 end
```

2.5 Local variable only used for Result

2.5.1 Rule

Description:

> *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.*

Listing 2.11 shows a feature populating a `LINKED_LIST` with some variable depending on the value of `a_boolean`. This is done by creating a local variable `l_list` of the type `LINKED_LIST` and, after adding the new element, assigning it to `Result` in order to return the list. This can be done in a much more efficient way by using `Result` directly to create the list and add the new element as we can see in Listing 2.17.

Listing 2.12: Example with no violation.

```
1  local
2    l_string: STRING
3  do
4    create l_string.make
5    create Result.make
6
7    l_string.append ("Hello ")
8    Result.append ("World")
9
10   Result := l_string
11 end
```

Listing 2.13: Attempted fix of example in Listing 2.12.

```
do
  create Result.make
  create Result.make

  Result.append ("Hello ")
  Result.append ("World")


end
```

This rule cannot be applied if the local variable which gets assigned to `Result` is also assigned to any other variable because then removing the local variable is not possible. Another case where we are unable to remove the local variable is when `Result` itself is created and or modified in the feature. An example of such

a case can be seen in Listings 2.12 and 2.13. If we would see this example as a violation of the rule and carry out the steps explained above we would change the semantics of the code. In the original code the returning `STRING` value would be `"Hello "` and after removing the local variable and replacing it with `Result` we would get `"Hello World"` as the output.

Another case where we are not allowed to remove the local variable is when the assignment to `Result` is nested. We cannot be sure if the assignment is even executed if it is in an if block for example. Therefore removing the assignment and the local variable could lead to changing the semantics of the code again. In Listings 2.14 and 2.15 we can see a code example of this case. In this example applying the fix to the code would lead to a different return value than in the original code. If `a_boolean`'s value is `False` then the original code would return `Void` and the fixed version always a string with value `"True"` (no matter the value of `a_boolean`.

2.5.2 Implementation

This rule is implemented using the visitor pattern. We check every `FEATURE_AS` node that has a return type as well as local variables. We then let the visitor run through the feature body and look for all the `ASSIGN_AS` and `CREATE_CREATION_AS` nodes. In the `CREATE_CREATION_AS` nodes we simply check if the target of the creation is `Result` and if so, we can be sure that there cannot be any violations because there is a `create Result` instruction of some form in the feature body.

Listing 2.14: Example with no violation.

```
1  local
2    l_string: STRING
3  do
4    create l_string.make
5    l_string.append ("True")
6
7    if a_boolean then
8      Result := l_string
9    end
10 end
```

Listing 2.15: Attempted fix of example in Listing 2.14.

```
do
  create Result.make
  Result.append ("True")

  if a_boolean then

  end
end
```

After checking the creation nodes we go to the assignment nodes. Here we check whether we have found an assignment from one of the local variables to `Result`. If there is one we remember the node and keep going through the feature body. In the case that we find another assignment to `Result` we have to abort and the rule is not violated.

After finishing the feature body and having found only one assignment to `Result` we check if the assignment node is not nested. When all these criteria are met we finally have a violation of the rule.

Listing 2.16: Example violation of local only used for Result rule.

```
1  local
2    l_list: LINKED_LIST
3  do
4    create l_list.make
5    if a_boolean then
6      l_list.extend (a_x)
7    else
8      l_list.extend (a_y)
9    end
10   Result := l_list
11 end
```

Listing 2.17: Automated fix applied to violation in Listing 2.16.

```
do
  create Result.make
  if a_boolean then
    Result.extend (a_x)
  else
    Result.extend (a_y)
  end

end
```

2.5.3 Fix

In order to fix violations of this rule we need to remove the declaration of the local variable that gets assigned to `Result`, replace every occurrence in the feature body of said local variable and finally remove the assignment to `Result` itself. Listings 2.16 and 2.17 show the application of this fix to the example introduced in Listing 2.11.

The EiffelStudio IDE already provides an implementation for removing an unused local variable automatically. So all we need to do is to run this feature, telling it that this local variable is unused and then its declaration will be removed.

2.6 Feature export can be restricted

2.6.1 Rule

Description:

> *An exported feature that is used only in unqualified calls may be changed to secret.*

This rule is about improving information hiding in the user's code [14]. In Listing 2.18 we can see an example violation of this rule. The feature `feature_one`'s export status is public which means that anyone can call it by either creating an instance of `A` or inheriting from `A` directly. Now if we inherit from `A`, we are allowed to call `feature_one` in an unqualified way as seen in class `B` from Listing 2.18. If these unqualified calls are the only calls to `feature_one` in the whole system, we can improve the information hiding by restricting `feature_one`'s export status to `{NONE}`. If there are any qualified calls to that feature though, we cannot restrict the export status because otherwise those feature calls could not be executed anymore.

Listing 2.18: Example violation of the feature export can be restricted rule.

```
class A                          class B

feature                          inherit
  feature_one                        A
    do
      do_nothing                 feature
    end                            feature_two
                                     do
end                                    feature_one
                                     end
                                 end
```

2.6.2 Implementation

We go through all the features of the class to be checked for violations and check two things for every feature. First, we check if the feature is called outside of the class' descendants. If we find a call to a feature from outside the class' hierarchy, we are not allowed to restrict this feature's export status since this call has to be a qualified one, otherwise we can continue. Secondly, we need to check if all the calls to the feature in the descendants are unqualified. One might think that this is automatically the case, but as Listing 2.19 shows, qualified calls are possible as well when an instance of the superclass is created. If we find such a qualified call we again cannot restrict the export status, even though the qualified call is in a class which inherits from the feature's class.

Listing 2.19: Example with qualified call in descendant class.

```
class B

inherit
  A

feature
  feature_two
    do
      instance.feature_one
    end

  instance: A

end
```

2.6.3 Fix

Since the export status of a feature is not explicitly given for every feature but rather for a whole block of features, it is difficult to predict how the user would want to arrange things when the violating feature is going to change its export status.

Listing 2.20: Generic parameters with more than one character.

```
1 class MY_HASH_MAP [KEY, VALUE]
```

One solution would have been to create a new `feature {NONE}` block at the end of all other feature blocks, but then again there could already be such a block somewhere else where all the secret features lie and the user would have to copy the feature back to that place himself.

Therefore we decided to omit an automatic fix for this rule and leave the user simply with the notification of the violation of the rule.

2.7 Generic parameter has more than one character

2.7.1 Rule

Description:

> *Names of formal generic parameters in generic class declarations should only have one character.*

According to the EiffelSoftware style guidelines for coding [3], formal generic parameters are only allowed to have a single character. Listing 2.20 shows an example of a generic class declaration that violates this rule.

2.7.2 Implementation

The implementation for this rule is simple: We check if the class contains generics via the feature `generics` from the `CLASS_AS` node. We then check if their names are longer than a single character. If so, we create a violation for this rule for this specific generic parameter.

2.7.3 Fix

We give the user several options for fixing this violation automatically. If a generic parameter is too long, we offer the user to replace the parameter and all its occurrences in the class by the first letter of said parameter. Now if there already is another generic parameter with only a single character and it happens to be the same as the first one of the violating parameter, we do not offer an automatic fix and the user must decide for himself which character he chooses. In the case that there are multiple parameters which are too long and all start with the same character, the user can choose from a dropdown menu which parameter should be replaced with its starting character automatically. The remaining parameter still must be changed by the user himself.

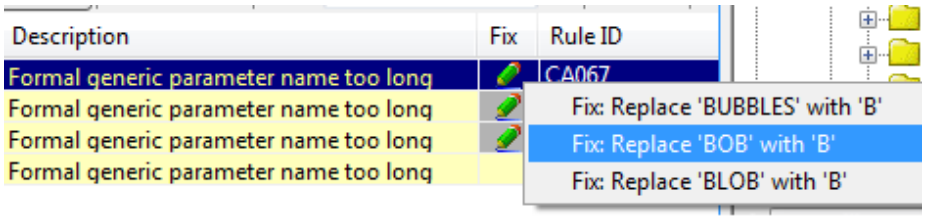

Figure 2.2: The dropdown menu for multiple fix options.

To achieve this we first iterate through all the generic parameters once to find the existing single characters and store them in a list. Now we iterate through the generic parameters a second time to find the ones which are longer than a single character. For every match, we check if its first character is already in the list and if it is we generate a violation without an automatic fix. If it is not in the list yet, we add a violation with an automatic fix which would replace this parameter with its first character. In the case that another match starts with the same character, we need to generate a new violation which contains all the fixes for this starting character, and also add the fix for this new match to the existing violations of the same character. This way we ensure that every generic parameter that starts with the same letter contains a dropdown menu of fixes for all the parameters that start with that letter. An example of this can be seen in Figure 2.2.

2.8 Mergeable conditionals

2.8.1 Rule

Description:

> *Successive conditional instructions with the same condition can be merged.*

Conditional statements, i.e. if-then-else blocks, which check for the same condition can be merged together to increase code readability. We only merge conditionals which are syntactically the same and whose conditions only consist of locals and boolean constants. This is because if we had a boolean feature in the conditions a parallel execution could change its value between the two blocks and therefore merging them would change the semantics of the code. Also to simplify the implementation we only allow the blocks to be successive in the code otherwise we would need to check all the statements in between to make sure the value of the condition does not change until execution reaches the second conditional.

Listing 2.21: Mergeable conditionals implementation.

```
1 if
2   attached {IF_AS} l_previous as l_if_1
3   and then attached {IF_AS} l_current as l_if_2
4   and then  l_if_1.condition = l_if_2.condition
5   and then is_condition_valid (l_if_1.condition)
6   and then is_body_valid (l_if_1)
7 then
8   create_violation(l_if_1, l_if_2)
9 end
```

2.8.2 Implementation

Again we use the visitor pattern over the abstract syntax tree. We visit all the `FEATURE_AS` nodes to check all the feature of the class. First we extract all the locals from the feature and save them in a list. Then we iterate through the feature body always looking at two successive instructions at a time. Listing 2.21 shows the next step of the implementation.[2] We see a large if block where all the conditions must hold. Let us look at them in more detail.

The variables `l_current` and `l_previous` hold the current and previous instruction of the feature body respectively. Lines 2 and 3 check if the instructions are of type `IF_AS` and if that is the case we assign them to the local variables `l_if_1` and `l_if_2`. The ordering matters here. `l_if_1` holds the conditional which is executed first in the code. We will see later why this is important.

Line 4 compares the two conditions of the if blocks to see if they are the same.

If that is the case line 5 checks whether the condition is valid. The feature `is_condition_valid` returns `True` when the condition of the if block only contains boolean constants and local variables. We only need to check one of the two conditions because we already know they are the same because of the check in line 4.

Finally, line 6 checks if the body of the condition that is executed first is valid. This is where the ordering of the two conditionals is important. The feature `is_body_valid` checks the body of the if block to see if any of the local variables used in the condition are assigned new values. If that is the case, it will return `False` and no violation will be created. Listings 2.22 and 2.23 show a short example why this check is important.

In this example if the conditionals are merged we change the value of `Result` at the end of the feature, even though the conditions of the two conditionals are the same. Because the first conditional's body changes the value of the condition, we cannot merge in this case. This is what the check in feature `is_body_valid` is for.

[2] N.B.: The code has been simplified to increase readability. E.g. the conditions in line 4 would need to be extracted further from the `IF_AS` node, but this notation is sufficient to get an understanding of the implementation of the rule.

Listing 2.22: Example with invalid body.

```
1  local
2    l_bool: BOOLEAN
3  do
4    l_bool := True
5    if l_bool then
6      l_bool := False
7      Result := 10
8    end
9    if l_bool then
10     Result := 12
11   end
12 end
```

Listing 2.23: Automated fix applied to violation in Listing 2.22.

```
local
  l_bool: BOOLEAN
do
  l_bool := True
  if l_bool then
    l_bool := False
    Result := 10
    Result := 12
  end


end
```

Going back to Listing 2.21, we are almost done. If all the checks return `True`, we can create a violation because the two conditionals can be merged without any changes to the outcome of the feature. So in line 8 we create a new violation for the two instructions of the feature body.

2.8.3 Fix

Automatic fixing for this rule is done by merging the bodies of the violating if blocks that share the same condition. Listings 2.24 and 2.25 show the application of this fix to an example of a violation of this rule.

Listing 2.24: Example violation of mergeable conditionals rule.

```
1  local
2    l_boolean: BOOLEAN
3  do
4    if l_boolean then
5      Result := 3
6    end
7
8    if l_boolean then
9      Result := Result + 5
10   end
11 end
```

Listing 2.25: Automated fix applied to violation in Listing 2.24.

```
local
  l_boolean: BOOLEAN
do
  if l_boolean then
    Result := 3
    Result := Result + 5




  end
end
```

Chapter 3

UI Improvements

This chapter describes the improvements in the user interface of the Eiffel Inspector and the new tags for class options.

3.1 Fix-Button

When we first started using the Eiffel Inspector, a violation could be fixed automatically by right clicking on it to bring up a menu that shows available fixes. If there is no fix, one still has to right click the violation to find out that that is the case. To remedy this design flaw we implemented a button which is active when there is a fix available and blank if there are no automated fixes for this rule violation.

In Figure 3.1 we can see a typical view of the Eiffel Inspector after the analysis with some violations in the list. We added a new column called "Fix" where the new button shows up. Upon clicking it, the usual drop-down menu appears to select a fix for the violation. An example of the drop-down menu is shown in Section 2.7 in Figure 2.2.

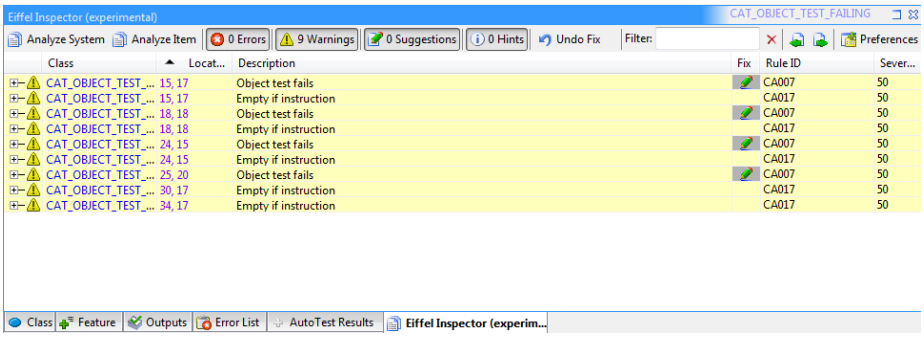

Figure 3.1: The Eiffel Inspector tool window.

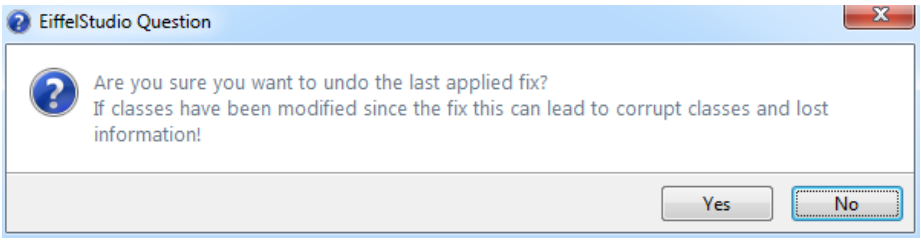

Figure 3.2: Warning for undoing fixes.

3.2 Undo-Button

Applying an automated fix for a violation can be fearsome because you do not know exactly how the Eiffel Inspector is going to adjust your code. Maybe it will change something you did not expect, or it will be changed in a way that does not suit your coding style. Either way, after an application of a fix, the code gets recompiled and analysed again, meaning that we are unable to just hit Ctrl + Z to undo the changes that have been made by the Eiffel Inspector.

Hence we decided to implement a new button in the user interface for reversing the changes made by a automated fix. When debugging a fix for a new rule you will want to be able to go back and forth without having to retype the violation again every time. We added this button to the top bar as we can see in Figure 3.1.

The button is greyed out when no fixes have been applied yet, and it also keeps a history of all the fixes that have been applied so far. So we can always restore the code back to the original state if needed.

One limitation though is that if the code is changed by the user, the undoing of fixes can result in an inconsistent state because statements might be in different locations or be gone altogether. That is why the user is presented with a warning when trying to undo a fix, letting him know of the potential mishap. An example of the warning can be seen in Figure 3.2

3.3 New class options

The Eiffel Inspector supports class options which can be added to the indexing clause (after the `note` keyword) of a class. One of these options is the tag `ca_ignoredby`. It means that this class should be ignored by a certain rule, and the tag should be followed by a rule ID or a list of such IDs. For better understanding we renamed this tag to `ca_ignore`.

While debugging and writing test cases for the new rules it is cumbersome to have so many violations showing in the Eiffel Inspector window when we are only interested in viewing the violations for the new rule we just implemented. Of course we could have used the `ca_ignore` tag and added all the rules except the one we wanted to see, but clearly that is a very tedious task given the

increasing number of rules, and every time we would implement a new rule, the list would have to be adjusted again.

That is why we introduced a new tag for these class options called `ca_only`. It works the same way, you put the tag into the indexing clause of your class and add a rule ID (e.g. `ca_only: "CA040"`, if you want to check rule #40). What this does, is that all the rules are now ignored, except the rule with the ID given from this tag. This also overwrites any restrictions from the tag `ca_ignore`. As soon as the `ca_only` tag is used, it is the only one that counts. This makes debugging rules much more user-friendly.

Chapter 4

Integration with the Verification Assistant

In this chapter we will discuss the integration of the Eiffel Inspector with the Verification Assistant tool.

4.1 The Verification Assistant

The Verification Assistant [17] is a tool implemented in the EVE IDE that brings together several tools. It provides a unified interface for analysis and verification tools to help developers maintain code quality and correctness. The implementation of EVE is freely available for download [7] and therefore continues to grow larger, offering integration for more verification tools. The currently available implementation supports the Eiffel Inspector and two other tools, namely AutoTest and AutoProof (See Chapter 1.2 for more information on these tools).

The Verification Assistant assigns scores to results of a testing tool for every routine tested. A score of -1 denotes certain incorrectness, for example when testing found an error. Assigning positive scores is harder because in most tools, we can never be 100% certain that a routine will never fail. So the positive score should be normalised so that it never exceeds an upper limit of 1, which denotes definite correctness and is therefore unattainable by testing.

For verification tools which are sound and complete, a successful test should generally yield a score of 1 and a failed proof a score of -1, since it denotes a certain fault. For incomplete tools, a failed proof simply denotes uncertainty and the scoring in that case depends on the details of the technique used in that tool.

Figure 4.1 depicts an example of an analysis showing the scores achieved from an analysis with Eiffel Inspector. For more readability, the scores in the user interface are adjusted by expanding their scale from [-1, 1] to [-100, 100], rounding to the closest integer.

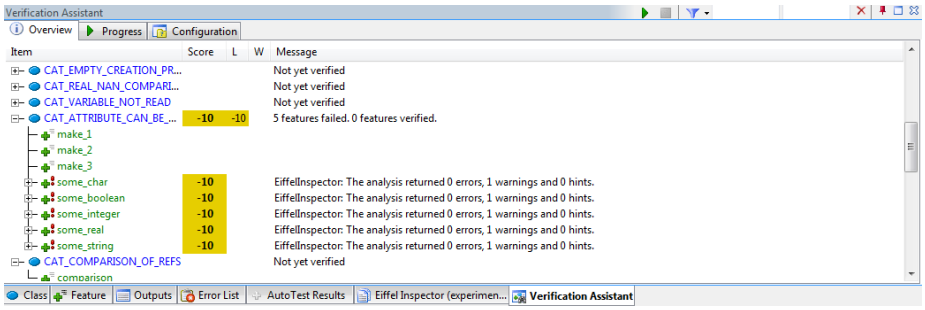

Figure 4.1: The Verification Assistant tool window.

4.2 The Eiffel Inspector

The Eiffel Inspector does not focus on correctness. Very few rules reveal definite coding errors and in most cases it is used for indicating bad code quality. When the tool yields many rule violations, it is still only little evidence for incorrectness and finding no violations does not show correctness of the program. The Verification Assistant accounts for this fact by setting the scores accordingly.

The rules in Eiffel Inspector are categorised as either a *hint*, a *warning* or an *error*. Hint rules are ignored entirely, as they are only indicating coding style errors. Error rules are very rare, and if violated the score is set to -1, denoting certain incorrectness. If only warning rules are violated the score still goes below 0 but only up to a maximum of -0.3, depending on the amount of violations of these rules.

Chapter 5

Improvements of existing rules

This chapter will list the improvements that have been added to the existing set of rules and fixes already implemented in the Eiffel Inspector framework. Some of the changes, like small refactorings or adjustments of code, have been very minor and are not listed here.

- CA_MOVE_INSTRUCTION_WITHIN_LOOP_FIX

  Rule #21 checks if there is a loop invariant computation that lies within a loop and therefore should be moved outside the loop. This rule supports an automated fix where the violating instruction was moved outside the loop. Another rule, #68, is addressing a similar issue where there is an object creation instruction that lies within a loop and should also be moved outside the loop but in this case it has to be before the loop. This rule also supports an automated fix, moving the instruction after the loop.

  We realised that the fixes for these rules are very similar and could be merged into one new fix called `CA_MOVE_INSTRUCTION_WITHIN_LOOP_FIX`. This automated fix takes three arguments: The `LOOP_AS` node of the loop with the violation, the `INSTRUCTION_AS` node of the violating instruction within the loop, and a `BOOLEAN` value, which defines if the violating instruction should be moved before or after the loop.

  This way both rules can use the same code, and maybe in the future even other rules can reuse the merged fix.

- CA_PRETTY_PRINTER

  In order to be able to have nicely formatted code when applying an automated fix of a rule violation, the class `CA_PRETTY_PRINTER` was introduced. This is an exact copy of the class `PRETTY_PRINTER` which can be used to output abstract syntax tree nodes in a well formatted manner. The copying was needed in order to make the class available for the code_analysis library.

One of the problems with the pretty printer was that if the user had formatted his code in a different way than the pretty printer would do it by default, then applying a fix would not only change the violating part, but in some cases also other parts of the code, if for example a whole if-block would be printed anew using the pretty printer. That is why we removed the pretty printer. As there was only one fix who was using the printer, we changed the implementation to calculate the indentation of the violating instructions first and then replace the text of the existing node, instead of creating a new one and sending it to the pretty printer.

- Automated fixes performance

  The way automated fixes were executed when the user would choose to run one was that the abstract syntax tree of the class under analysis would be visited in its entirety until the violating node has been found and the necessary changes could be made. In most cases though, the fix already holds a reference to the violating node and would be able to directly change that node without having to run the (in most cases) lengthy process of visiting all the nodes until the correct one is found.

  To correct this design flaw we introduced a new feature for the class `CA_FIX` (every fix inherits from this class), called `execute`. This feature contains the usual code of visiting all the class' nodes so that the old way of fixing violations is still usable. Now if a fix does already hold a reference to the violating node, one can simply redefine the `execute` feature and only make the changes to that node. This will increase the performance of most of the fixes, as visiting all the nodes becomes redundant in those cases.

- Folder hierarchy

  The folder hierarchy in the code_analysis library was becoming increasingly confusing because of the constantly growing amount of rules and fixes implemented for the Eiffel Inspector. We redesigned the folder hierarchy making it clean and sensible.

Chapter 6

Conclusions and future work

6.1 Conclusions

In this thesis we implemented eight new rules for the rule-based framework Eiffel Inspector. For all but one of these rules we also implemented an automated fix that resolves rule violations. Most of the rules use a visitor pattern and iterate through certain abstract syntax tree nodes created by the Eiffel compiler. In the end we tested the rules against two large library classes EiffelBase [6] and EiffelVision2 [5] to ensure the rules are running correctly.

We also improved the user interface of the Eiffel Inspector. Implementing and debugging new rules or fixes has been simplified. Developers can now undo fixes they applied to their code with a new button and also choose to run the Eiffel Inspector with only a subset of the rules by using the new class tag `ca_only`.

With these improvements to the Eiffel Inspector developers have an even better way of finding code quality issues in their projects and save a lot of time compared to checking for code quality with a document or some other guideline. Because the framework is rule-based and easily extendable and the implementation of EVE is freely available for download [7], avid programmers can easily create new rules for their own needs or edit existing ones.

The Eiffel Inspector is overall well designed and also adding new rules is very well documented [2]. The only thing that is not well documented is the user interface classes for the framework.

6.2 Future Work

As discussed about in Chapter 5 one of the new fixes implemented is called `CA_MOVE_INSTRUCTION_WITHIN_LOOP_FIX`. This fix moves an instruction from within a loop to the outside of the loop, either in front or directly after it. In order to

keep the code layout clean we need to know the indentation of the loop instruction because the loop in question might be nested in another construct. This indentation is currently being calculated with a lot of nested calls to `substring` and `index_of` on the raw text of the `LOOP_AS` node. This could be changed by either introducing a feature `indentation` for the abstract syntax tree nodes or by adding something like the `PRETTY_PRINTER` for the Eiffel Inspector.

The `PRETTY_PRINTER`, or rather a copy of it called `CA_PRETTY_PRINTER`, was already in use when we first started with the Eiffel Inspector, but the problem mentioned above still remains the same. The pretty printer does print the successive instructions with correct indentations but the initial indentation is still always put to zero and in fact, cannot even be changed (in the current implementation the `CA_PRETTY_PRINTER` was removed, see Chapter 5 for more details).

Another problem in the Eiffel Inspector framework is the way the code analysis works with the class tags. We introduced the `ca_only` class tag to reduce the time an analysis would take by a considerable amount by only checking one (or several) of the rules instead of all of them. To our surprise we found that the analysis still took the same time. We later realised why this is the case: The tags used in the indexing clause are only applied to the rule set after the analysis is already done and the results are being displayed. Meaning that no matter how little rules the user chooses, the analysis will still run the full set of rules to then only display the violations of the rules that the user has chosen.

As of now all the new rules, fixes, improvements, and the user interface changes are only implemented in the EVE IDE, which is a research branch of EiffelStudio, the main IDE for Eiffel. All changes to the Eiffel Inspector version of EVE need to be transferred to the version in EiffelStudio.

Bibliography

[1] 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.

[2] EiffelSoftware contributors. EiffelSoftware Open Source - CA Adding new rules. https://dev.eiffel.com/CA_Adding_New_Rules, March 2015.

[3] EiffelSoftware contributors. EiffelSoftware Open Source - Style Guidelines. https://dev.eiffel.com/Style_Guidelines#Letter_case, March 2015.

[4] 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.

[5] Eiffel Documentation. EiffelVision 2 Library. https://docs.eiffel.com/book/solutions/eiffelvision-2, March 2015.

[6] Eiffel Documentation. The EiffelBase Library. https://docs.eiffel.com/book/solutions/eiffelbase, March 2015.

[7] Eiffel Verification Environment (EVE). http://se.inf.ethz.ch/research/eve/, March 2015.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] Robert Green and Henry Ledgard. Coding guidelines: Finding the art in the science. *Queue*, 9(11):10:10–10:22, November 2011.

[10] 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.

[11] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[12] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *IEEE Software*, pages 22–24, 2009.

[13] Microsoft. FxCop. http://www.microsoft.com/en-us/download/details.aspx?id=6544, March 2015.

[14] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[15] Pmd. http://pmd.sourceforge.net/, March 2015.

[16] SonarQube™. http://www.sonarqube.org/, March 2015.

[17] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, SEFM '11. Springer, 2011.

[18] 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.

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

[20] Stefan Zurfluh. *Rule-based Code Analysis*. ETH-Zürich, 2014.