Chair of Software Engineering

AutoTeach: Incremental Hints For Programming Exercises

Master Thesis

Paolo Antonucci
ETH Zurich
apaolo@student.ethz.ch

March 17, 2014 - September 17, 2014

Supervised by:
Dr. Marco Piccioni
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

In recent years, e-learning platforms and MOOCs (Massive Open Online Courses) have gained great popularity, and the Computer Science area is not excluded from this trend. A challenge for MOOCs is providing effective exercises to the students, and this is even more important in applied disciplines such as programming. A major difficulty of this challenge is ensuring that students do not get stuck in solving the exercises, which eventually leads to giving up. This is one of the most significant drawbacks of MOOCs over more traditional forms of teaching, where teachers can provide direct guidance to students who lack the necessary idea for solving an exercise.

We designed AutoTeach, an incremental hint system for programming exercises. Given a complete solution of a programming exercise, AutoTeach can generate several stripped out versions of it, numbered by *hint level*, which initially only contain the skeleton of the solution's code, and then, for higher hint levels, gradually reveal more parts of the code and/or hints on how to write it. Students who get stuck on an exercise can ask for a hint (one or multiple times) and receive versions of the code where more and more parts of the solution are revealed, until they are able to proceed on their own. This way we ensure that students who happen to lack the necessary idea for solving an exercise can receive the help they need, instead of immediately giving up.

Acknowledgments

I would like to thank very much my supervisor, Marco Piccioni, for continuously supporting me throughout all phases of this thesis. Many thanks to Prof. Bertrand Meyer for his support and the great interest shown in the project.

I thank the whole group of the Chair of Software Engineering for reviewing my code and providing abundant and extremely useful feedback. I am also grateful to the Eiffel Software developers, in particular Emmanuel Stapf, for the support I received when working on Eiffel Inspector and Eweasel.

Many thanks to my friend Alessandra, who was a great listener during the development of the ideas of this thesis, helping me refining them as I presented them to her. She even managed not to look too bored. Finally, special thanks to my father, who was also a great listener, and spent several hours proof-reading my final report instead of sleeping.

Paolo Antonucci

Contents

Chapter 1

Introduction

Recent years have seen a constantly increasing interest in modern teaching and learning methods. Devices such as tablets and e-book readers have become more and more popular with students and teachers. E-learning platforms, with particular regard to MOOCs (Massive Open Online Courses), are gaining a strong momentum, both in mass e-learning websites and in academic environments, where they are often used as a complement to more traditional forms of didactics, taking the form of SPOCs (Small Private Online Courses, [3]).

While these new platforms offer several benefits compared to traditional teaching methods, and work very well in replacing one-way lectures, they face a major challenge when it comes to providing students with exercises and assessing their solutions. Many MOOCs try to avoid the problem as much as possible by resorting to closed-answer (e.g. multiple choice) tasks. These are of course very easy to correct (often automatically), but are generally not sufficient for training and testing students on all the skills they are expected to acquire when studying a discipline. This is also and especially the case with Computer Science and programming-related courses, where learning without real practice is not reasonable. This is the scenario this thesis tackles.

1.1 Automated exercise assessment

Arguably the first challenge to be faced when offering open-answer programming tasks is their assessment. There is a number of properties that can be assessed in such tasks, such as

- **syntactical and semantic validity:** does the code actually compile?
- **correctness:** does the code do what it is supposed to do in all possible cases?
- **efficiency:** is the code efficient and does it scale up well?
- **code quality:** does the code respect general good practices and conventions?

- **respect of additional constraints:** does the code satisfy the additional constraints given by the exercise, if any? For example, in an exercise about recursion, did the student really implement a recursive solution?
- **originality:** did the student solve the exercise by herself or did she copy the code from somewhere?

Currently, in most MOOCs and programming exams, some of these points are addressed by automated assessment systems. Most of the times, the program submitted by the student is compiled (validity test) and run against a set of predefined inputs in order to check its correctness in several cases. The execution time is usually limited, so that inefficient solutions get rejected. There are also a number of tools that can check students' code for similarities in order to prevent plagiarism, some of them having existed for as long as 20 years so far [2].

What is not yet always receiving proper consideration is the evaluation of code quality. This aspect is very important not only for grading the students' code, but also for providing the students with feedback on how to improve an exercise that is not going to be graded.

1.2 Guiding students towards the solution

Besides evaluation and automatic generation of feedback, another major challenge provided by open-answer exercises is helping students make their way to the solution when they are stuck and cannot proceed. In traditional forms of teaching, one could always ask the professor or a teaching assistant for a hint on how to proceed or what approach to take, but in fully automated forms of teaching this is not possible.

A feasible alternative to the direct interaction between teachers and students would be to prepare in advance a list of hints to be provided to students upon request. Even though these hints might still not be sufficient to answer specific questions by the students, they still prove helpful in many cases where students don't know where to start at all or what approach they should take. Ideally, hints should be incremental: if a student asks for help, she is initially only given a hazy clue of the solution, which will still not "spoil" it. If she keeps asking for more help, she will receive increasingly detailed hints.

Unfortunately this problem tends to be very often ignored in MOOCs (in those that provide open-answer exercises at all), and automatic, progressive hint systems are still uncommon.

Unfortunately, even MOOCs that offer open-answer exercises tend to ignore the problem of students getting stuck and being unable to proceed, and automatic, progressive hint systems are still not common.

1.3 Our contribution

In this thesis we present AutoTeach, an incremental hint system for programming exercises in Eiffel. AutoTeach is a processing tool that takes complete solutions to programming exercises as input, and generates stripped-out versions of the solutions, where initially only the skeleton of the code is visible. For each input file, several versions can be generated, with increasingly large parts of the code shown. AutoTeach is capable of running in a fully automated mode, where a default smart processing policy is applied to the exercise, or in a manual mode, where the teacher can control and customize to her like the visibility of any part of the code and add textual hints to the code. There is no sharp separation between the two approaches: nothing prevents the teacher from running AutoTeach in automatic mode on an exercise and make slight adjustments to its behavior or complement the automatic output with textual hints.

In the following chapters we will thoroughly explain the concepts on which AutoTeach is based, and make some practical considerations about its use in teaching.

In addition to AutoTeach, this thesis also contributes a set of code analysis rules for Eiffel Inspector [17]. Eiffel Inspector is a code analysis tool for Eiffel, which has the potential of being used for providing students with automated feedback on the quality of their code. While it is a general purpose code analysis tool, many of the rules it supports, including those that we present here, can be useful to students, in that they can catch mistakes that most experienced programmers would never make.

Finally, our contribution includes the implementation of support for testing Eiffel Inspector rules in Eweasel [10], a unit testing tool for the Eiffel compiler. This included some changes to Eiffel Inspector itself, mainly consisting of improving its support for the command line, with the goal of making it easier to support unit-testing of rules and taking it closer to being fit for being used for the analysis of students' code.

1.4 Outline

In the following chapters, we will present and discuss AutoTeach. After an initial quick tour in chapter 2, chapter 3 will thoroughly discuss all the concepts on which AutoTeach is built. In chapter 4 we will then focus on the usage of AutoTeach by teachers and on practical tips. Chapter 5 will then discuss the most important aspects of the implementation of AutoTeach.

We will complete our tour with some final conclusions in chapter 6.

At the end of the thesis, appendix A contains reference information that would not have fit well into the regular chapters.

In appendix B, the new rules for Eiffel Inspector, developed as a part of this thesis, are presented and documented.

Finally, appendix C consists of a short report on the extension of the Eweasel tool and the changes made to Eiffel Inspector itself in order to support unit-testing of code analysis rules.

Chapter 2

AutoTeach Overview

In this chapter we provide a first general introduction to AutoTeach. We go through the evolution of the concept, from the initial idea to the final implementation. In doing this, we show some first examples, which will be helpful for starting the next chapter with an initial idea of what should be expected.

2.1 Introduction

AutoTeach is a tool for generating incremental hints for programming exercises written in Eiffel. The main purpose for which it was developed is to use it in the 2014 edition of the first year Introduction to Programming course of the Computer Science department at ETH Zurich. Since the 2013 edition, the course offers a MOOC platform complementing traditional didactics. The MOOC platform includes programming exercises that can be solved directly from within the web browser. The solutions submitted by the students are compiled and unit-tested over a set of inputs, and the students receive an automatic feedback on how their code performed in the tests. AutoTeach will be integrated with this module, so that students who get stuck and cannot proceed in solving an exercise can click a button and request one or more hints.

AutoTeach works by taking a complete solution to a programming exercise and generating several stripped out versions of it, where parts of the code are hidden (optionally replaced by placeholders) and textual hints, if defined by the teacher, are inserted. These versions are numbered from 0 up. Numbers represent **hint levels**: the higher the level, the more detailed the code and the textual hints are.

The idea of providing incremental textual hints, spread over several hint levels in order to reduce spoilers, is not new, and is quite common in other contexts, such as hintbooks for adventure videogames [12]. Here however we take it one step further, by accompanying or even replacing textual hints with the automatic revealing of some parts of the code.

In the following paragraphs we will try to give a first idea of how this works

in practice. We will use the following code fragment as our hello-world example.

Listing 2.1: *Eiffel*: Our hello-world example

```
sum (a_numbers: ARRAY [INTEGER]): INTEGER
    -- Sum of all elements in 'a_numbers'.
  do
    Result := 0

    across a_numbers as ic loop
      Result := Result + ic.item
    end
  end
```

2.1 actually shows the **solution** of our exercise: in our example, students will be required to implement feature *sum*.

2.2 Evolution of the concept

The initial idea was to focus on textual hints and have two separate modes for AutoTeach: the "annotated mode", where the source code would be annotated with textual hints by the teacher, and the "unannotated mode", to be used where the teacher did not provide textual hints.

2.2.1 Annotated mode

The annotated mode was meant for those exercises which had been annotated by the teacher with textual hints, organized in increasing levels, and meant to be shown to the students. Only the skeleton of the features to be implemented should have been visible to the students, where skeleton usually means the declaration, with arguments and return types, and the local declarations. In our example, the solution's code could have been annotated the following way:

Listing 2.2: *Eiffel*: Hello world example with textual hints annotations

```
sum (a_numbers: ARRAY [INTEGER]): INTEGER
    -- Sum of all elements in 'a_numbers'.
  do
    --# [1] HINT: First, start by initializing Result to zero.
    --#-  This is not necessary, but makes the idea clearer.
    Result := 0

    --# [2] HINT: Maybe you need a loop.
    --#-  Perhaps you should iterate on 'a_numbers'.
    across a_numbers as ic loop
      --# [3] HINT: In every iteration you should add the current
            item to the current result.
      Result := Result + ic.item
    end
  end
```

Comments starting with "`--\# [1] HINT`" are special annotations indicating a textual hint which should be visible at hint level l or greater.

Listing 2.3 shows within a single listing the produced output at hint level 1, 2, and 3.

Listing 2.3: *Eiffel*: Output resulting from listing 2.2

```
-- Hint level 1.
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      --# [1] HINT: First, start by initializing Result to zero.
      --#-  This is not necessary, but makes the idea clearer.

      -- Your code here!

    end

-- Hint level 2.
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      --# [1] HINT: First, start by initializing Result to zero.
      --#-  This is not necessary, but makes the idea clearer.

      -- Your code here!


      --# [2] HINT: Maybe you need a loop.
      --#-  Perhaps you should iterate on 'a_numbers'.

      -- Your code here!

    end

-- Hint level 3
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      --# [1] HINT: First, start by initializing Result to zero.
      --#-  This is not necessary, but makes the idea clearer.

      -- Your code here!


      --# [2] HINT: Maybe you need a loop.
      --#-  Perhaps you should iterate on 'a_numbers'.

      -- Your code here!

      --# [3] HINT: In every iteration you should add the current
          item to the current result.

      -- Your code here!

    end
```

As we see, the teacher writes the hints directly within the code, in the form of special comments. Even though the code is hidden, the teacher can (and should) still write each hint at the appropriate location within the code. This makes it very easy for her to write hints and also ensures that code placeholders (`-- Your code here!`) are properly interwoven with textual hints. The student can trust code placeholders and be sure, for instance, that no code is to be

inserted before the first hint.

2.2.2 Unannotated mode

The so-called unannotated mode was intended to work in those cases where the teacher did not provide any annotation in the source code. It would work by revealing the structure of the code gradually. In our example, the input would be the original, unmodified listing 2.1, and the output would look like this:

Listing 2.4: *Eiffel*: Output resulting from listing 2.1

```
-- Hint level 1.
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do

      -- Your code here!

    end

-- Hint level 2.
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do

      -- Your code here!

      across a_numbers as ic loop

        -- Your code here!

      end
    end

-- Hint level 3.
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      Result := 0

      across a_numbers as ic loop

        -- Your code here!

      end
    end

-- Hint level 4. (solution)
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      Result := 0

      across a_numbers as ic loop
        Result := Result + ic.item
      end
    end
```

As we see, students only start by seeing the empty skeleton of the routine. Then, if they ask for help, they first get to see the existence of a loop, without showing its body, then instructions outside any compound statement (in this case, the initial assignment to *Result*), and only finally they see the complete code. In the full *from* form of Eiffel loops, the loop initialization instructions, termination condition, invariant and variant are also shown at a later time than the existence of the loop itself.

The behavior described above is defined by a policy, which specifies which parts of the code should be visible at different hint levels (e.g. contracts become visible at level 3). A default policy exists, so that there is no need to specify the policy for every single exercise. Actually, the behavior described above is the one implied by the default policy.

In the initial idea, this mode should have been used as a fallback in the case of unannotated exercises, however, after the final visibility model for the parts of the code had been developed, it turned out to be more effective than expected.

2.3 The final solution

Smart defaults are good, but flexibility is even better. A design goal of AutoTeach was to make its behavior as customizable as possible, so that the teacher could change anything she didn't like. This was achieved in two ways:

- **Full custom policies:** the teacher can specify a custom policy and completely redefine what parts of the code must be shown at what level.
- **Processing directives within the code:** the teacher can write some special annotations, under the form of special comments, that override the default behavior of AutoTeach in that file, or even in a single specific occurrence of a code block.

In planning this, it became soon evident that there was no real reason why textual hints and automatic or semi-automatic processing should stay separate, and that joining them would have enabled teachers to write more effective hints, without prejudice to the possibility of relying on AutoTeach's defaults and let it do its job without intervention when that flexibility is not needed.

The mixture of the two modes makes hints like the following possible:

Listing 2.5: *Eiffel*: Output obtained by combining textual hints and the gradual revealing of the code structures

```
-- Hint level 1.
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      --# [1] HINT: First, start by initializing Result to zero.
      --#- This is not necessary, but makes the idea clearer.
      Result := 0
```

```
      -- Your code here!

    end

-- Hint level 2.
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      --# [1] HINT: First, start by initializing Result to zero.
      --#-  This is not necessary, but makes the idea clearer.
      Result := 0

      --# [2] HINT: Maybe you need a loop.

      -- Your code here!

    end

-- Hint level 3.
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      --# [1] HINT: First, start by initializing Result to zero.
      --#-  This is not necessary, but makes the idea clearer.
      Result := 0

      --# [2] HINT: Maybe you need a loop.
      --# [3] HINT: Perhaps you should iterate on 'a_numbers', like
          this:
      across a_numbers as ic loop
        --# [3] HINT: Now, in every iteration you should
        --#-  add the current item to the current result...

        -- Your code here!

      end
    end

-- Hint level 4. (solution)
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      --# [1] HINT: First, start by initializing Result to zero.
      --#-  This is not necessary, but makes the idea clearer.
      Result := 0

      --# [2] HINT: Maybe you need a loop.
      --# [3] HINT: Perhaps you should iterate on 'a_numbers', like
          this:
      across a_numbers as ic loop
        --# [3] HINT: Now, in every iteration you should
        --#-  add the current item to the current result...
        --# [4] HINT: ...like this:
        Result := Result + ic.item
      end
    end
```

These "hybrid hints" come to the price of adding some additional processing directives to the code, but the result is more effective. Listing 2.6 shows how the input code was annotated for obtaining this result. Notice the use of the

'SHOW_NEXT' and 'SHOW_NEXT_CONTENT' directives.

Listing 2.6: *Eiffel*: Annotated hybrid version of the sample exercise

```
sum (a_numbers: ARRAY [INTEGER]): INTEGER
    -- Sum of all elements in 'a_numbers'.
  do
    --# [1] HINT: First, start by initializing Result to zero.
    --#-  This is not necessary, but makes the idea clearer.
    --# [1] SHOW_NEXT instruction
    Result := 0

    --# [2] HINT: Maybe you need a loop.
    --# [3] HINT: Perhaps you should iterate on 'a_numbers', like
        this:
    --# [3] SHOW_NEXT loop
    --# [4] SHOW_NEXT_CONTENT loop
    across a_numbers as ic loop
      --# [3] HINT: Now, in every iteration you should
      --#-  add the current item to the current result...
      --# [4] HINT: ...like this:
      Result := Result + ic.item
    end
  end
```

Our quick tour is complete. In the next chapter we will define more precisely the concepts on which AutoTeach is built and we will dig in deeper detail into its usage.

Chapter 3

AutoTeach principles

In this chapter we discuss what model and what concepts have been developed and how they are used in AutoTeach. While, strictly speaking, this chapter contains all the notions necessary for making full use of AutoTeach, it is mostly focused on the theoretical aspect. Some practical tips and good practices for using AutoTeach will be presented in chapter 4. Finally, implementation details are presented in chapter 5.

3.1 Trileans

A key concept for Autoteach, one that needs to be specified, is that of "trileans". This is defined and described in this section.

3.1.1 Definition

The three-valued boolean, or "trilean" as we call it here, is a type. Instances of this type can assume three values: *true*, *false* and *undefined*. From now on we will use the word "trilean" to indicate any instance of the trilean type, which in the context of programming generally means a variable or a value.

We say that a trilean is **defined** when its value is either *true* or *false*. Trilean can be regarded as a supertype of boolean: any boolean variable can be seen as a trilean which is always defined.

In three-valued logic, the *undefined* value often means that the truth value is unknown, thus it is often also called *unknown* or *undetermined*. In the context of AutoTeach, the most appropriate term to express what *undefined* means is "not set". We will be using trileans for options and flags that can be explicitly set to *true* or *false*, or can be left undefined if not set.

There is flourishing literature about three-valued logic (also known as ternary logic), see [16] for an accessible starting point. Here, we will only define one operation on trileans, which will be useful later on.

3.1.2 Subjection operation

The "subjection" operation represents a trilean being overridden by another one. The result of the operation will always be the value of the overriding trilean (right operand) unless it is undefined, in which case the result is the value of the first trilean (left operand).

More formally, we define the **subjection** operation as follows.

$$\mathbf{A}\ \textit{subjected to}\ \mathbf{B} = \begin{cases} \mathbf{B} & \text{if } \mathbf{B} \text{ is defined} \\ \mathbf{A} & \text{otherwise} \end{cases}$$

This yields the following truth table:

A \ **B**	**True**	**Undefined**	**False**
True	True	True	False
Undefined	True	Undefined	False
False	True	False	False

Table 3.1: Truth table for the trilean subjection operation.

It is important to notice that the subjection operation is not commutative. In fact, we can define the inverse operation, which we call **imposition**, as follows:

$$\mathbf{A}\ \textit{imposed on}\ \mathbf{B} = \begin{cases} \mathbf{A} & \text{if } \mathbf{A} \text{ is defined} \\ \mathbf{B} & \text{otherwise} \end{cases}$$

This operation yields the following truth table:

A \ **B**	**True**	**Undefined**	**False**
True	True	True	True
Undefined	True	Undefined	False
False	False	False	False

Table 3.2: Truth table for the trilean imposition operation.

3.2 AutoTeach basics

3.2.1 Input and output

AutoTeach is a command line tool integrated with the Eiffel compiler. A graphical user interface is currently under development, however it is not a part of this thesis.

The main parameters required by AutoTeach are:

- An Eiffel project. The project must be complete and valid, i.e. it must compile without errors.
- The name of one or more classes in the project which should be processed.
- A natural number, or a range of natural numbers, specifying the range of hint levels to run AutoTeach with.

AutoTeach will first compile the code. If the compilation is successful, the specified classes are processed. The source text of each class is scanned sequentially, with a single pass, and the output is generated on the fly. The output is a stripped out copy of the input source code, where certain sections and parts of the code are hidden (and possibly replaced by placeholders).

AutoTeach requires the input classes to be compiled. By default, the Eiffel compiler will ignore classes that are not referenced (directly or indirectly) by the root class of the application. If this situation arises, AutoTeach prints a warning and skips the classes that have not been compiled.

It is also important for the input code to be well-formed, including formatting and indentation. In the output, AutoTeach will keep the original formatting and indentation whenever possible, meaning that incorrectly indented input code will result in incorrectly indented output code (GIGO principle [14]). The code must be indented with tabs, not spaces, as this is the convention in Eiffel.

The output for every class is saved to a file, the location of which must be specified by the arguments. The process is repeated for all the selected hint levels.

It is worth mentioning explicitly that AutoTeach doesn't need to be invoked on the spot every time a student asks for a hint. In fact, all exercises can be processed in advance, for all relevant hint levels, and students requesting a hint can simply be served the right file.

3.2.2 Hint levels

Hint levels are execution modes identified by natural numbers. The higher the hint level, the more detailed hints are generated for the students. Conventionally, hint level 0 is used for generating the skeleton of the code provided to students as a starting point for solving an exercise. What is generated is the input of the exercise, it is provided to all students and it is not regarded as a hint. A hint level of 1 or higher will contain hints and clues which will be only given to students who ask for help by pressing a button in the GUI.

The maximum hint level is not fixed: depending on the AutoTeach configuration, the behavior will be defined up to a certain level n. It is technically possible to run AutoTeach at higher levels than n, however the generated output will be the same as for level n.

3.2.3 Code blocks

An important choice in designing AutoTeach was the granularity, that is, the elementary units which it should be able to handle. With respect to this, we defined a list of supported code "blocks". Every block type has a name, which we will also use when referring to block types in this thesis. Blocks are classified as **atomic blocks** and **complex blocks**. The difference between the two categories is that complex blocks may contain other blocks (whether atomic or complex) nested in them, whereas atomic blocks cannot contain any other kind of block. The exhaustive list of block types, including a short description for each of them, is found in the appendix (A.2).

Examples of *atomic* blocks are:

- ***instruction***: instructions of all kinds, with the exclusion of compound statements (such as 'if's and loops).
- ***assertion***: assertions in contracts, including class invariants, and in loop invariants.
- ***if_condition***: the condition of an 'if' statement.

This list, albeit incomplete, gives a first idea of the minimum processing granularity: instructions and assertions are generally the smallest processing unit handled by AutoTeach. It is not possible to reason about smaller program building blocks, such as single operators or keywords and, in general, single expressions. Boolean or integer expressions can however constitute a block in some specific cases (such as blocks of type *if_condition*).

Examples of *complex* blocks are:

- ***precondition***: the precondition of a routine, including the 'require' keyword.
- ***if***: an "if" statement.
- ***if_branch***: any of the two branches of an "if" statement, including the "then" or "else" keyword.

Figure 3.1 (on the next page) shows a code sample and the composition of blocks within it.

3.2.4 Hiding code

Generally, whenever a part of the code is hidden, a blank line is inserted. By default, a placeholder (a comment prompting the students to fill in some code) is also inserted.

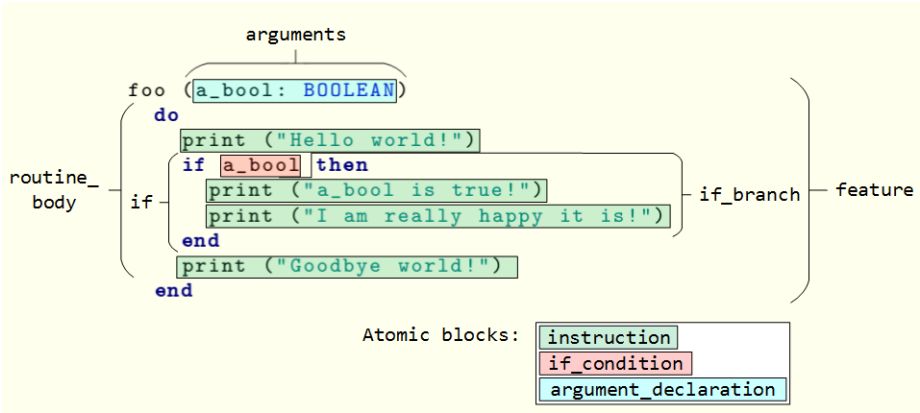

Figure 3.1: Code example showing the composition of code blocks. Complex blocks are indicated by braces, atomic blocks by colored rectangles.

Regardless of how extended the hidden code region is, only one placeholder (or blank line) is inserted, so that students cannot infer the number of missing instructions from the number of blank lines. This is also true if several consecutive independent blocks are hidden.

Even when hiding some parts of the code, AutoTeach does its best to produce a syntactically valid program and alter the input abstract syntax tree as little as possible. A challenge posed in this by the Eiffel language is that it exclusively supports line comments: comments can only be terminated by a newline character. TThis makes it difficult to replace with an inline placeholder a hidden inline block such as routine arguments or the condition of an *if* statement, which we would like to replace with an inline placeholder. In the majority of cases this is solved by inserting a new line before and after the section to be hidden, so that the placeholder can be alone on a line. However, in the case of condition of *if*s, the solution is to insert a fake (but syntactically valid) condition as an inline placeholder, as inserting additional lines in these cases would have resulted in particularly bad-looking code.

3.3 Visibility

In the first two paragraphs of this section we will introduce two alternative, orthogonal approaches for determining which code blocks should be visible in the output. In the third paragraph we will show how these two approaches can be combined, and how the resulting model is more powerful than the two approaches taken singularly. These three paragraphs are perhaps the most crucial of the whole thesis.

3.3.1 Block visibility

As previously said (3.2.1), AutoTeach works by processing classes sequentially. In the previous section, we learned that the source code of classes, in the abstraction used by AutoTeach, is composed of blocks. Clearly, the ultimate question we are interested in answering for every occurrence of a code block, is "should we print this block or should we hide it?".

The question is answered by a lookup in the **visibility table**. The visibility table is a table associating block types and hint levels. Every row of the visibility table represents a block type, every column represents a hint level, and every cell contains a trilean value indicating whether or not the corresponding block type should be visible, and thus be printed to the output, at that level. Although cells contain trilean values, let's assume for now that all values are defined, so that they are in fact boolean values. The following example shows a simplified visibility table:

Hint level / Block type	**0**	**1**	**2**
feature	True	True	True
if	False	True	True
instruction	False	False	True

Table 3.3: Simplified example of a visibility table.

Whenever a block is encountered, a simple lookup in this table tells whether that block should be printed to the output at the current hint level or not. At any time, AutoTeach always has an active visibility table. More details on the default visibility tables and how they can be customized will be provided in a later section (3.4).

In the following sections we will sometimes refer to this concept as "**basic visibility**" when this is necessary for disambiguation.

Let's see an example. Consider the same code fragment of figure 3.1, which we report here again as a listing:

Listing 3.1: *Eiffel*: Example

```
1  foo (a_bool: BOOLEAN)
2    do
3      print ("Hello world!")
4      if a_bool then
5        print ("a_bool is true!")
6        print ("I am really happy it is!")
7      end
8      print ("Goodbye world!")
9    end
```

As a visibility table, we will use table 3.3. Of course this table is incomplete, as the entries for several blocks highlighted in figure 3.1 are missing, but it will suffice for this simplified example. Processing listing 3.1 with table 3.3 at hint levels from 0 to 2, would generate the following output:

Listing 3.2: *Eiffel*: Complete output produced by processing listing 3.1 with visibility table table 3.3.

```
-- Hint level 0.
  foo (a_bool: BOOLEAN)
    do

      -- Your code here!

    end

-- Hint level 1.
  foo (a_bool: BOOLEAN)
    do

      -- Your code here!

      if a_bool then

        -- Your code here!

      end

      -- Your code here!

    end

-- Hint level 2.
  foo (a_bool: BOOLEAN)
    do
      print ("Hello world!")
      if a_bool then
        print ("a_bool is true!")
        print ("I am really happy it is!")
      end
      print ("Goodbye world!")
    end
```

Note that the output consists of a separate file for each hint level, we have merged here the output for different levels into one listing for the sake of conciseness.

3.3.2 Complex blocks and content visibility

The approach shown in the previous section is very simple and quite flexible, yet not flexible enough. For example, consider again listing 3.1. Assume that, on a certain hint level, the teacher wants to show all the instructions appearing directly within the body of a routine, but wants to hide all those inside the body of an *if* statement, thus she would like to have the following output:

Listing 3.3: *Eiffel*: Desired output

```
  foo (a_bool: BOOLEAN)
    do
      print ("Hello world!")
      if a_bool then

        -- Your code here!
```

```
    end
    print ("Goodbye world!")
end
```

For what has been explained so far, now there is no way to achieve this. If we set the *instruction* visibility to *true*, then all the instructions will be shown, including those within *if* blocks. Similarly, if we set it to *false*, then none of them will be shown. We need a new paradigm to make it possible. This new paradigm, which we call **content visibility**, is presented hereafter.

This new paradigm is orthogonal to, and independent of, the mechanism that we have presented in section 3.3.1 (basic visibility), so the reader is encouraged to forget the mechanism of basic visibility for a while and restart from scratch. Only at a later stage (section 3.3.3) we will put together the two paradigms and show how they combine.

Every point in the code, where "point" means any place in the text on which we could click and set the cursor for editing, has a **content visibility policy** in force in that place. The content visibility policy is a trilean value which, as an approximative definition, indicates whether or not the code appearing in that region should be visibile (i.e. printed to the output). Note that we said a trilean, which means it can also be undefined.

In our case, to achieve the desired goal, we need to ensure that the content visibility policy be *false* within the body of the *if* instruction and *true* outside of it.

In section 3.2.3 we discussed the distinction between atomic blocks and complex blocks. This distinction becomes relevant here, as complex blocks can specify a **content visibility policy**, which is valid within their body. We establish that if the policy for a block is *undefined*, then it is inherited from the parent block. In case of a root-level block, which has no parent, if its content visibility policy is undefined, then it remains undefined.

The content visibility policy for the complex block types is defined in the **content visibility table**. This table is totally analogous to the basic visibility table, in that every row represents a complex block type and every column a hint level, and the value in every cell, which is a trilean, indicates the content visibility policy for that block type on that hint level. A simple example is the following:

Hint level / **Block type**	**0**	**1**	**2**
routine_body	False	True	True
if	False	False	True

Table 3.4: Simplified example of a content visibility table.

As in the previous section (3.3.1), this table is simplified and many entries are missing, for the sake of simplicity, but it is sufficient for our example.

Going back to our example (3.1), our problem is solved by table 3.4 at hint level 1. At level 1, blocks of type "*routine_body*" (the *do ... end* block spanning from line 2 to 9) have a content visibility of *true*. This means that at lines 3 and 8 the applicable content visibility is *true*, therefore the instructions on those two lines get printed. On the contrary, blocks of type "*if*" have a content visibility of *false*, therefore at lines 5 and 6 the applicable content visibility is *false* and the two instructions on those lines will not be printed.

The complete output produced by listing 3.1 processed with content visibility table 3.4 is shown in the following listing:

Listing 3.4: *Eiffel*: Complete output produced by processing listing 3.1 with content visibility table 3.4.

```
-- Hint level 0.
  foo (a_bool: BOOLEAN)
    do

      -- Your code here!

      if a_bool then

        -- Your code here!

      end

      -- Your code here!

    end

-- Hint level 1.
  foo (a_bool: BOOLEAN)
    do
      print ("Hello world!")
      if a_bool then

        -- Your code here!

      end
      print ("Goodbye world!")
    end

-- Hint level 2.
  foo (a_bool: BOOLEAN)
    do
      print ("Hello world!")
      if a_bool then
        print ("a_bool is true!")
        print ("I am really happy it is!")
      end
      print ("Goodbye world!")
    end
```

We can now define more precisely the rules governing content visibility:

- Every point in the code has an applicable content visibility. In particular:
  - The content visibility policy at any point which is not contained in any complex block is *undefined*.

  - The content visibility policy at any point which is directly or indirectly contained in a complex block is equal to the content visibility of the innermost complex block in which that point is contained.

- The effective visibility of a code block is affected by the content visibility policy applicable at its position[1] the following way:
  - **Atomic blocks**: their visibility is defined by the content visibility policy in force at their location. If the content visibility policy at their location is *true*, they will be printed to the output, if it is *false* they will be hidden.
  - **Complex blocks**: **no effect**. Complex blocks are immune to the content visibility policy in force at their location. This design choice, which might be perplexing, is motivated discussed in section 3.8.3.

As we see, the mechanism of content visibility and the associated content visibility table make it possible to hide or show atomic blocks depending on their location in the code and not on their type. This is exactly the contrary of the basic visibility, where the visibility of a block is determined by the type of the block and not by its position in the source code. Here, blocks of the same type (in our example, *instruction*) are handled differently depending on where they are located. Also, this mechanism always preserves the visibility of the complex blocks, which are the "boning" of the code, only allowing to hide the atomic blocks that they contain, which are the "flesh". This is often desirable, as it gives the students clues about how the code should be structured without revealing the exact implementation.

The concept of content visibility alone is simple and powerful, but it is not flexible enough. While revealing the structure of the code can be desirable at some times, we definitely need the ability to completely hide at least some types of complex blocks, such as *if* instructions. In the next section we will see how we can solve this and other problems by putting the notion of content visibility together with the one of basic visibility which we encountered in section 3.3.1.

3.3.3 The whole picture

In sections 3.3.1 and 3.3.2 we have introduced the concept of blocks **basic visibility** and 3.3.2 respectively. We have seen how these two notions are orthogonal, they can be explained and make sense independently of each other.

Also, a careful reader will recall that, both in the basic and in the content visibility table, we said that the table cells contain **trilean values**. This means that both the basic visibility of a block and the content visibility of a complex block will not always assume a defined value. We were a bit vague on the reasons for this and on what happens if the visibility of a block is undefined.

[1] Note that it is not important to specify whether we mean the content visibility at the beginning of the code block or at its end. As the content visibility policy solely depends on the parent complex block, it is guaranteed to be the same both immediately before the beginning and right after the end of our block.

It is now time to combine the two visibility paradigms, describe how we can take advantage of these trilean values and understand the full picture of how visibility works with AutoTeach.

Briefly, blocks in AutoTeach are subject both to basic visibility and content visibility. The effective visibility of a block is the result of a combination of both notions. More precisely:

- The visibility of **complex blocks** is simply equal to their basic visibility, as the content visibility policy is never applicable to complex blocks. However, while immune to it, complex blocks have the ability to *define* a content visibility policy valid within their body.
- The visibility of **atomic blocks** is equal to their basic visibility, *if it is defined*. If it is undefined, then their effective visibility *is equal to the content visibility in force at their location*. In other words, the effective visibility of an atomic block is *its basic visibility* ***imposed on*** *(3.1.2) the applicable content visibility at that location.*

As both notions are used simultaneously, we need both the basic visibility table and the content visibility table. These tables can be used harmoniously together only if they are thought from the beginning as the two sides of the same coin. From now on, we will use the term **hint table** to refer to the combination of both tables.

This is the complete picture. This solution provides the answer to several open questions:

- *Why do we use trileans instead of booleans for both basic and content visibility?* Because we want to make it possible to leave one of them undefined and let the other do the job. For example, in cases like example 3.1 we might want to leave the basic visibility of blocks of type *instruction* undefined, so that their effective visibility is decided by the content visibility in force at the locations where they appear. In addition, trileans for the content visibility allow a nested block to inherit the visibility policy from its parent block.
- *How do we hide complex blocks of a certain type?* By setting the basic visibility of that block type to *false*. This is the only way, as this cannot be achieved by working with the content visibility.
- *Why does basic visibility have priority over complex visibility in defining the visibility of an atomic block?* Because we consider it to be more specific. In fact, basic visibility always refers to a specific block type, while content visibility is valid within a region of the code which may contain heterogeneous blocks. In general, it is a design principle of AutoTeach that more specific policies are stronger than general policies (this concept will be stressed later on, in section 3.6.3).
- *What happens if a complex block has an undefined basic visibility? Will it be shown or not? Also, what happens if an atomic block has an undefined basic visibility and the applicable content visibility is also undefined?*

In these cases AutoTeach would not know whether that block should be printed or not.

The answer to this question is: **this should never happen!** A well-designed hint table should always ensure that this situation can never happen. For example, if a hint table sets the basic visibility of *instruction* blocks to *undefined*, then it must ensure that the content visibility of every possible region where *instruction* blocks can occur is always either *true* or *false*.

If this undesirable situation eventually happens, AutoTeach will not abort the processing, and will print the offending block out of courtesy, as if its visibility were *true*, but it will issue a warning to the user, informing her that she is using a malformed hint table.

The "orphan rule"

There is one additional rule complementing what we have seen until now and governing the visibility of blocks. AutoTeach is a didactic tool, and, as such, aims at being helpful to students and never confuse them. We believe that altering the structure of the syntax tree of the code would be inelegant and misleading for students. For this reason, we established the following principle:

Regardless of its theoretical visibility, no block can ever be printed to the output unless its direct parent is printed as well.

In fact, if we were to allow that, we could end up printing orphan nodes, that are "detached" in the output syntax tree. This would often break the syntax of the language, and, what is worse, would be extremely confusing, as it would make it possible to, for example, hide an if statement, but print the instructions in the 'then' block as if they were not inside a compound statement. This is something we decided to disallow. We refer to this principle as the "orphan rule". We will see that the sole exception to this rule are textual hints (3.5.3), which are however not a part of the syntax tree of the original code. Apart from this, there are no other exceptions or ways to bypass this rule.

Visibility of comments

Comments, in most if not all programming languages, are considered a break in the abstract syntax tree of the input source code. Eiffel is no exception to this: comments appear inside breaks. Not all breaks include comments, break can also consist of whitespace and return characters.

As comments can appear at any location, it would have been very difficult to come up with an accurate policy for processing them and deciding which ones should make into the output and which not.

AutoTeach's policy for processing comments is the following:

- Generally, after a block has been processed, the break immediately following it is processed as if it were a part of that block. For example, if an

assignment instruction is followed by ten blank lines, in turn followed by an *if* instruction, the ten blank lines will be processed as if they were a part of the assignment instruction, not the *if*.

- When a comment is processed, it is printed to the output if
  - The block of which it is considered part is visible (basic visibility) **AND**
  - The applicable content visibility for that region **is not** *False*. If the visibility in the region is *undefined*, the comment will be shown.

Note that these rules do not have solid foundations, they should be regarded as a "best effort" policy, which may not be optimal for all situations. However, in cases where it is important that a comment is shown or hidden at a specific hint level, an easy solution is to convert that comment to a textual hint.

3.4 Hint tables

As we said in section 3.3.3, we use the term "hint table" to indicate the combination of a basic visibility table and a content visibility table designed to work together. AutoTeach comes with two different built-in hint tables, used with different purposes (more on this in 3.4.2), and with the ability of loading a custom hint table.

3.4.1 Compactness

AutoTeach supports 24 different block types, 17 of which are complex blocks. This means that a complete hint table would have to contain 41 total rows (24 in the visibility table and 17 in the content visibility table). Every row would need to have as many cells as the number of supported hint levels. In the case of the default table for automatic mode, which has eight levels, the hint table would end up having 328 cells, which is a quite high number. This would make it very complicated to maintain the default tables and, more importantly, to write custom hint tables. For these reason, we have designed two simplifications that can make hint tables more compact and manageable.

The first possible simplification is that *any row can be omitted*. If a row is omitted, then every attempt to access that row at any hint level will always return *undefined*. This makes it optional to include the rows for which one never (i.e. at no hint level) wants to specify a visibility (or content visibility) other than *undefined*.

The second simplification is that *the rows that are specified do not have to be complete*. In fact, there is no concept such as "complete rows", as hint tables do not explicitly specify a maximum hint levels. We could be more precise by simply saying that rows are not required to have all the same length. The hint level grows from the left to the right: any attempt to access a row for a hint level that exceeds the row length returns the value of the row at the highest defined hint level.

```
# Hint level:                   0       1       2       3
            feature               T       T       T       T
content     feature               U       U       U       U

            arguments             T       T       T       T
content     arguments             F       T       T       T

            precondition          T       T       T       T
content     precondition          F       F       T       T

            locals                T       T       T       T
content     locals                F       F       F       T
```

Figure 3.2: Screenshot from a text editor showing a portion of the default auto hint table in complete form.

```
# Hint level:                   0       1       2       3
            feature               T
content     feature               U

            arguments             T
content     arguments             F       T

            precondition          T
content     precondition          F       F       T

            locals                T
content     locals                F       F       F       T
```

Figure 3.3: Screenshot from a text editor showing the same table as figure 3.2, this time in compact form. Notice how easier it is to understand what the table does at a glance.

The effect of these simplifications can be appreciated by comparing figures 3.2 and 3.3. These show screenshots taken from a text editor showing a portion of the default hint table for automatic mode. The second one is in compact form (in theory, the second row is superfluous and could have been removed altogether). The two tables are perfectly equivalent, however notice how the second form is much easier to read and understand.

3.4.2 Modes

AutoTeach uses the term "mode" for referring to the hint table being used. We say that AutoTeach is running in "automatic mode" (or just "auto") when the default hint table for automatic mode is being used, in "manual mode" when the default table for manual mode is active, and in "custom mode" when a user-defined hint table is in use. We use the word 'mode' for convenience, but modes do not carry any more meaning than this.

The running mode can be specified with a command line switch. The switch is optional, if it is omitted then the default mode is "auto". It is also possible to switch mode on the fly at any time while processing an exercise using a special annotation in the source code of the exercise (this is later explained in section 3.5.3).

3.4.3 Hint table for automatic mode

One of the two pre-loaded tables is the table for the so-called "automatic mode". This table is the "smart" table which teachers should use when they do not plan to customize the way exercises are processed and they trust the default AutoTeach behaviour to do the job. It aims at providing a general-purpose sensible policy, which in most cases will be just fine. This is also the default table.

The policy implemented by this table is summarized here. Please bear in mind that in future versions of AutoTeach it might change.

The following list is incremental. It is implicit that whatever is shown at level 'x', remans visibile at levels greater than 'x'.

- **Level 0:** only show the skeleton of features. In AutoTeach, this means showing:
  - name of the feature
  - return type (for queries)
  - feature comment

  Show the existence, but hide all the content, of the following blocks:
  - routine arguments
  - precondition
  - locals
  - body (*do/once ...*)
  - postcondition
  - class invariant
- **Level 1:** show all routine arguments.
- **Level 2:** show all contracts: preconditions, postconditions, class invariants.

- **Level 3:** show all local declarations.
- **Level 4:** show the existence of compound statements.
  - Show the existence of *if* statements, but hide their conditions and their bodies.
  - Show the existence of *inspect* statements, including the value being inspected and all branches. Keep hiding the body of branches.
  - Show the existence of all clauses that are actually present in the input loop (with clauses we mean *from*, *invariant*, *until*, *loop* and *variant*). Hide the content of all these clauses.
- **Level 5:** show the secondary clauses of compound statements, keep hiding their bodies.
  - Show the conditions of *if* statements, keep hiding the body of their 'then' and 'else' branches.
  - Treat *inspect* blocks the same as in level 4.
  - Show the content of loop initialization (*from*), loop invariant (*invariant*), loop termination condition (*until*), and loop variant (*variant*). Keep hiding the content of loop bodies (*loop* clause).
- **Level 6:** show all instructions in routines which are *not* inside any compound statements (*if*, *inspect*, loops). Keep hiding the content of the bodies of compound statements.
- **Level 7:** show everything.

The full source of the default hint table for automatic mode is found in appendix A (A.6).

3.4.4 Hint table for manual mode

AutoTeach includes a second built-in table which is associated to manual mode. This table is extremely simple, only consisting of one level.

- **Level 0:** show the existence of all features. Of every feature, show the name, arguments, type (for queries) and feature comments. Show the existence of all feature clauses (*require*, *local do/once*, *ensure*), hide the content of all feature clauses.

This table is clearly of little use as it is. In fact, it is provided as a template for those cases where the teacher wants to completely customize the way an exercise is processed. Selecting manual mode is a way for the teacher of saying "Give me a blank canvas on which I can paint my own picture". How a teacher can customize the processing of a single exercise will be made clear in a few pages.

The full source of the default hint table for manual mode is found in appendix A (A.7).

3.4.5 Custom hint tables

AutoTeach supports loading a custom hint table from a text file. This is done by providing the path of the text file as a command line argument when executing AutoTeach. The syntax of this text file is presented in the Appendix.

This feature is optimal for those cases where a batch of exercises should be processed automatically (or almost automatically), however the teacher would like to use a different policy than the default one for the whole batch.

Loading a custom hint table when launching AutoTeach makes it available for use during AutoTeach's execution, however the table will only be activated if custom mode is selected. This is done either by selecting custom mode from the command line or with a special annotation in the source code requesting to switch to custom mode. It is possible to switch between 'auto', 'manual' and 'custom' mode at any time of the processing and as many times as desired.

Loading multiple custom hint tables and switching between them within the same exercise is not supported.

3.5 Meta-commands

So far, we have vaguely mentioned "special annotations" in AutoTeach's input source code, without properly defining them. AutoTeach supports indeed special annotations which alter the way the code is processed. These processing directives are called **meta-commands** [2]. In this section we will define the syntax and characteristics of meta-commands and introduce the most common of them.

3.5.1 Syntax

Meta-commands are written under the form of comments, so that they do not affect the semantics of the input programs. A meta-command can only start at the beginning of a comment and terminates at the end of that comment, which, in Eiffel, means at the end of the line. Meta-commands are not allowed to span multiple lines.

Comments containing a meta-command must start with a hash (#) symbol following the double dash which opens the comments. Whitespace is allowed before and after the hash symbol. The hash is followed by the keyword of the command. All commands require some kind of payload following the command keyword, the structure of which depends on the specific command.

It is possible (but not required) to specify a range of hint levels before the command keyword. If this is done, then the command will be evaluated or executed *only if AutoTeach is running at a level within the specified range*, otherwise, the command will be ignored. All this translates to the following general syntax:

[2] The prefix 'meta' is used to prevent confusion with the normal Eiffel instructions.

```
--# [x-y] COMMAND_KEYWORD <payload>
```

'x' and 'y' are natural numbers. 'x' is the minimum hint level for the command to be valid, 'y' is the maximum level. Both of them are optional, meaning that "[x-]" is a valid level specification, which can be translated as "from level x on". Similarly, "[-y]" is also a valid level specification, meaning "up to level y". It is also possible to omit both parameters at the same time and write "[-]". This is translated to "on any level", and is equivalent to entirely omitting the level specification.

It is also allowed to provide one number only, i.e. "[x]". However, this does not mean "exactly on level x", but "from level x on", that is, it is equivalent to writing "[x-]". The reason for this apparently arbitrary choice is that most meta-commands for which a level is specified are in fact commands causing some clue (textual hints or parts of the code) to be shown to the student. Generally, once a clue is shown, it will not be hidden again at higher hint levels, so it is useful to have a compact way for indicating that a certain clue should be shown *from level 'x' on*. In the rare cases where a command should be executed on a single specific hint level, this is obtained by specifying the same number both as minimum and maximum hint level, i.e. by writing "[x-x]".

For improved readability, especially in the case of textual hints (3.5.3), an optional colon is allowed immediately after the command keyword:

```
--# COMMAND_KEYWORD: <payload>
```

Commands working on blocks

Many commands require the name of a block type to be provided as a payload, for example

```
--# SHOW_NEXT instruction
```

For these commands it is always possible to provide multiple block types on a single line, separated by commas and/or whitespace, like this:

```
--# SHOW_NEXT instruction, if, if_condition
```

This is perfectly equivalent to writing the following:

```
--# SHOW_NEXT instruction
--# SHOW_NEXT if
--# SHOW_NEXT if_condition
```

The original command will be expanded to these three sub-commands, which will be executed separately. If one of them turns out to be invalid (for example because of an incorrect block type name), the others are nevertheless accepted.

The full BNF-form grammar specifying meta-commands is found in the appendix. (A.3)

3.5.2 General characteristics of meta-commands

The first thing to say about meta-commands is that they never get printed to the output. The entire line containing a meta-command is suppressed. This is true even for incorrect commands. AutoTeach treats every comment starting with a hash as a meta-command, and even if one such line cannot be parsed AutoTeach will issue a warning, but still not print the line to output.

The only exception to this is for command 'HINT', the purpose of which is exactly to print a hint to the output.

Meta-commands are always processed regardless of the visibility of the code surrounding them.

As we already said, as a single Eiffel comment may not span across multiple lines, commands may not either. In addition to this, it is highly recommended to place all meta-commands on their own lines, i.e. *not* on the same line of some part of the code. Failure to do so may result in incorrect indenting or whitespace (including newlines) in the output.

Last but not least, meta-commands only affect the class they appear in. At the end of the processing of any class, the effects of any active meta-command cease.

3.5.3 Supported meta-commands

In this section we list all the supported meta-commands that do *not* work with blocks. Meta-commands working with blocks require an ampler treatment and will be presented in sections 3.6 and 3.7. An exhaustive list of all meta-commands is found in the appendix (A.4).

For sake of simplicity, we do not specify hint level bounds for meta-commands in this section, however we remind the reader that these can *always* be specified.

MODE

```
--# MODE newmode
```

This command switches to the specified mode. Valid values for 'newmode' are 'auto', 'manual' and 'custom'. As a reminder, as soon as the end of the class is reached, AutoTeach will revert to the default mode for that run.

PLACEHOLDER

```
--# PLACEHOLDER (on | off | True | False)
```

This command enables ('on', 'True') or disables ('off', 'False') the insertion of a code placeholder when a region of code is skipped. Even when placeholders are disabled, a blank line is always inserted (with the exception of some inline blocks).

HINT and hint continuation

```
--# [4] HINT: In the loop body, you should check
--#-  if the current item is present in the hash table.
--#-  If not, you should add it and increase the counter.
```

The 'HINT' meta-command is used for defining a textual hint. If the hint level is within the specified range, the entire line will be printed to the output. This is the only case where a meta-command is printed directly to the output.

It is particularly important to specify a range of hint levels. If no levels are specified, the hint will always be visible. Remember that when a single hint level is specified, that level is interpreted as a lower bound, e.g. the command in the above example will be printed when the exercise is processed on a hint level *greater than or equal to* 4.

This is also the only exception to the "orphan rule" (3.3.3). Hints can be printed even if they are located in a region which is being hidden. In this case, AutoTeach will attempt to re-adjust their indentation so that it matches the indentation of the output, preventing students from inferring how deep a hint was nested by checking its indentation.

We said that meta-commands are not allowed to span across multiple lines. However, this constraint is particularly troublesome in the case of hints, as it is often necessary to write hint text which is too long to fit on a line. A workaround could be spanning the text of a single hint across multiple 'HINT' commands, one per line, however this is not very convenient. This is the reason why the **hint continuation command** was introduced.

A hint continuation is indicated by a single dash after the meta-command hash, as in the example above. The dash must be followed by at least one space or tab character. If this is done, the line is treated as a continuation of the previous hint, and will be shown on the same hint levels which were specified for the previous hint. If AutoTeach finds a hint continuation command and the last encountered meta-command was not a hint, the line is ignored and a warning is issued.

Meta-comments

```
--## We will never show the next instruction to students,
--## otherwise it's too easy.
```

It is possible to insert meta-comments. Meta-comments are comments referring to the processing of the exercise by AutoTeach or to other meta-commands, not to the code itself. The practical difference between normal comments and meta-comments is that meta-comments are always filtered away and never appear in the output.

Technically, any line identified as a meta-command will be filtered from the output, however AutoTeach will issue a warning for every invalid meta-command which it encounters. Meta-comments of course do not trigger any warning.

Meta-comments are inserted by starting a comment with a double hash instead of a single one. Note that internally meta-comments are implemented as a normal command, where the command keyword is '#'. This means that a space (or a tab) is mandatory following the second hash, otherwise AutoTeach will not recognize it as the command keyword.

3.6 Visibility overriding

In section 3.3 we explained the notions of basic visibility and content visibility. For both of them, we said that they are controlled by policies which are defined in two tables, which, considered jointly, we call 'hint table'. We then saw how these notions are put together by AutoTeach.

We have mentioned several times, and in particular when we presented the manual mode, that teachers have the possibility to fully customize the way an exercise is processed, beyond what the use of a custom hint table would allow. Now that we have introduced meta-commands, in this and in the following section we can finally show how this can be done.

We will take the same approach we used when we presented the concepts of basic and content visibility and consider the two notions separately in the following two sections. Then in section 3.6.3 we will put all together and show the full flexibility achieved by AutoTeach.

3.6.1 Basic visibility overriding

Let's start from the basic visibility. Please temporarily forget the notion of content visibility: assume that all blocks, atomic or complex, are born equal and that their visibility is determined by a single lookup in the basic visibility table.

Now, imagine that the behavior implied by the basic visibility default table be satisfactory, except for some very small detail. For example, you would like all instructions to shown at level 5 already. What you could do is to use a custom hint table, but it would not be very convenient do have to fill a custom table just for changing one value. The same effect can be achieved more conveniently by using a **global visibility override**.

Doing a global visibility override means instructing AutoTeach so that, within a certain class, all blocks of a certain type have a certain visibility defined by the user with a meta-command instead of the default one expressed by the table. This is achieved through the use of the following meta-commands:

```
--# SHOW_ALL <block_type>
--# HIDE_ALL <block_type>
--# RESET_ALL <block_type>
```

Needless to say, as for all meta-commands, a hint level range can be specified between the hash symbol and the keyword using syntax '`[x-y]`'. If this is done, the meta-command will only be processed when AutoTeach is running at a hint level within the specified range.

The 'SHOW_ALL' meta-command signifies that the visibility of all blocks of type 'block_type' should be overridden to *true*. The 'HIDE_ALL' command overrides it to *false*. The 'RESET_ALL' command restores the visibility to its default value (the one defined in the table).

Each of these commands is valid starting from the location where it appears, and remains valid until another command of the same type specifies something different or until the end of the class is reached. The reason why these commands are called *global* override commands is that they apply to all blocks of a certain type, and not just one specific instance.

In our example, where we want routine arguments to be always shown, this could be achieved by writing '`--# [5] SHOW_ALL instruction`' at the beginning of our class.

We can achieve a finer granularity control than this by using **local visibility overrides**. This is done with the following meta-commands:

```
--# SHOW_NEXT <block_type>
--# HIDE_NEXT <block_type>
```

Unsurprisingly, these commands cause the next occurrence of a block of type 'block_type' to be shown or hidden, taking priority over anything else, that is, default visibility and any applicable global visibility override.

An insight on how this is implemented and handled by AutoTeach will prove helpful when, in the next sections, we put together basic and content visibility overriding.

For every block type, AutoTeach keeps a **block visibility descriptor**. An example of a descriptor is shown in figure 3.4. This descriptor contains three trilean attributes:

- The default visibility for that block type[3].
- The current global visibility override for that block type (undefined if no global override is in force).
- The current local visibility override to be applied to the next encountered instance of that block type (also undefined if no local override is in force).

The 'SHOW_ALL', 'HIDE_ALL' and 'RESET_ALL' commands set the global visibility override flag of the specified block type respectively to *true*, *false* and *undefined*. The flag is automatically reset to *undefined* at the end of the processing of every class.

Similarly, the 'SHOW_NEXT' and 'HIDE_NEXT' commands set the local visibility override flag of the specified block type to *true* and *false* respectively.

[3]Technically, the default visibility is not an attribute, but a query which will fetch the value from the active visibility table, but thinking of it as an attribute is simpler and makes no practical difference for the sake of comprehension.

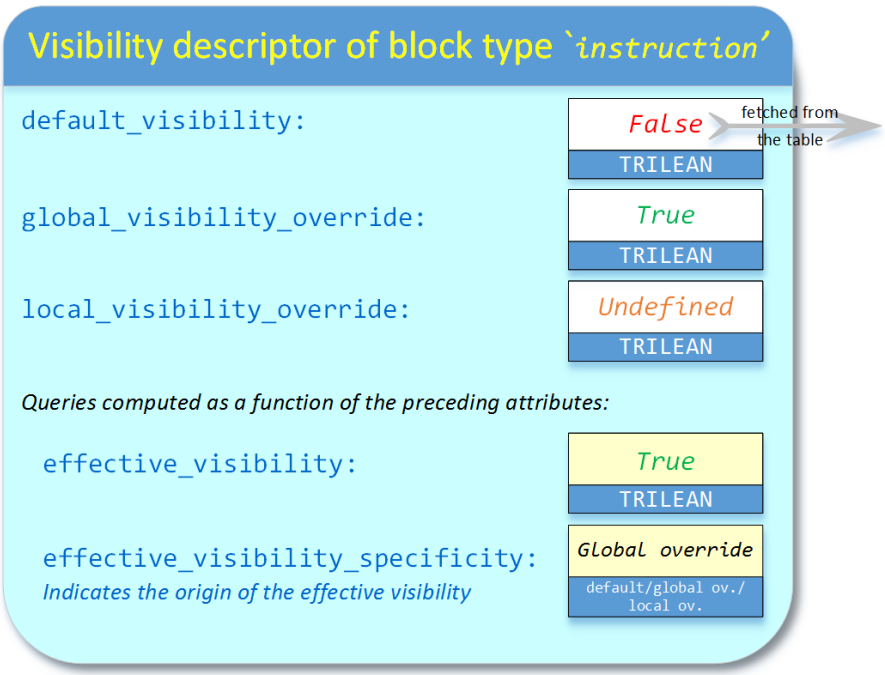

Figure 3.4: Example of a block visibility descriptor.

The flag is reset to *undefined* as soon as an instance of that block type is encountered and processed (or at the end of the processing of the current class, if it comes first).

The block type descriptor also provides two queries. The first of them (*effective_visibility*) calculates the effective visibility of an instance of that block type under the current status. This, remembering the order of priorities and that the three attributes are all trileans, can be written as:

default_visibility ***subjected to*** *global_visibility_override* ***subjected to*** *local_visibility_override*

In the example of figure 3.4, the result is *true*, as "*false* ***subjected to*** *true* ***subjected to*** *undefined*" yields *true*.

The second query, *effective_visibility_specificity*, tells us where the effective visibility value came actually from. The answer can be one of *default visibility*, *global override* and *local override*. This information basically means how specific (and hence how "strong") the visibility value of that block is. If a block has been made visibile (or hidden) thanks to a local visibility override command ('SHOW_NEXT'), there should be nothing able to reverse that, and we will see that, for ensuring this, it is very useful to have a query answering the question "how specifically, therefore strongly, has the visibility of this block type been defined?". From now on, we will use the term '**specificity**' to refer to this concept.

In the example of figure 3.4, the result of the second query is "*global override*", as the effective visibility value comes from the global override flag.

3.6.2 Content visibility overriding

Let's now forget about basic visibility and switch to content visibility. As we remember, it is possible to specify a content visibility for complex blocks, which is valid inside their body and which will affect the visibility of atomic blocks (and atomic blocks only) contained in them. This can be used to specify, for example, that all instructions within a routine body should be visibile, but all instructions nested in an *if* statement should be hidden.

As in the case of basic visibility, being limited to modifying the content visibility table is a bit restraining. Very often it is desirable to ask AutoTeach to fully show or hide the content of a specific complex block (such as an *if* statement). This can be easily achieved by overriding the content visibility.

The idea of content visibility override is exactly the same as basic visibility overriding. The only differences derive directly from the differences between the concepts of basic visibility and content visibility. Most notably, content visibility can only be defined for complex blocks.

The commands for overriding the content visibility of a certain (complex) block type are analogous to those for basic visibility overriding. The following commands are to be used when a global content visibility override is desired:

```
--# SHOW_ALL_CONTENT <complex_block_type>
--# HIDE_ALL_CONTENT <complex_block_type>
--# RESET_ALL_CONTENT <complex_block_type>
```

As for basic visibility, the override is reset when the end of the current class is reached.

For a local visibility override, the following commands are used:

```
--# SHOW_NEXT_CONTENT <complex_block_type>
--# HIDE_NEXT_CONTENT <complex_block_type>
```

As for basic visibility, as soon as the next block of type '¡complex_block_type¿' is encountered and processed, the local override is reset.

In addition to their basic visibility descriptor, every complex block type also has a **content visibility descriptor**, as shown in figure 3.5.

This descriptor is totally analogous to the basic visibility descriptor, the functioning of which has been thoroughly explained in the previous section (3.6.1). We would only like to stress two things here:

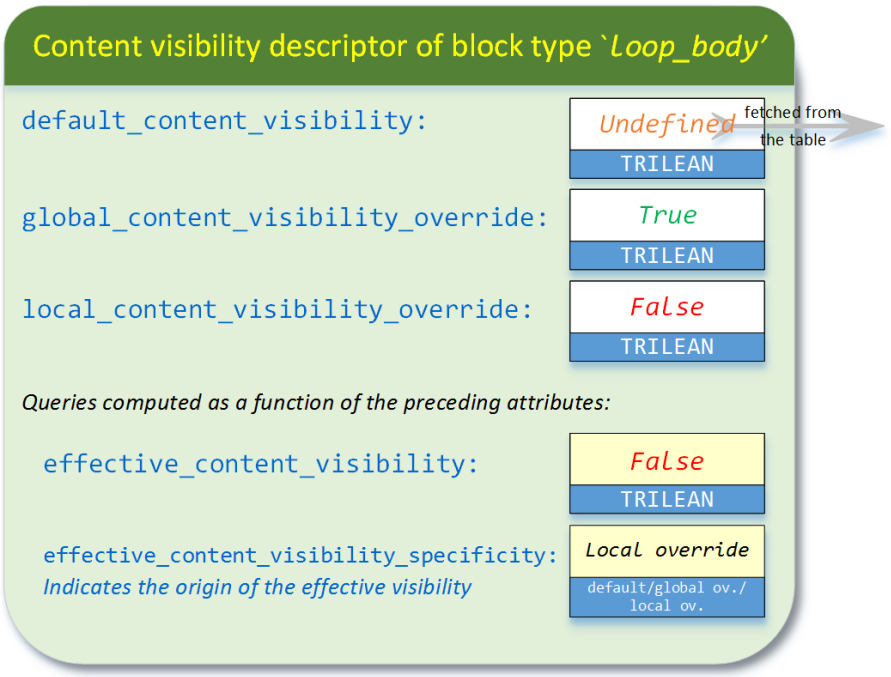

Figure 3.5: Example of a block content visibility descriptor.

- Complex blocks types have *both* a basic visibility descriptor *and* a complex visibility descriptor.[4]
- The visibility of a complex block *is not affected* by its content visibility descriptor. That will only define what content visibility policy will be applicable *inside* the block, which will affect the visibility of atomic blocks contained in it.

The content visibility descriptor also has a query returning the specificity of the content visibility policy. As in the case of basic visibility, this query tells us if the content visibility valid within a complex block simply comes from the content visibility table (default, unoverridden value), from a global override, which is stronger, or from a local override, which is even stronger.

3.6.3 Putting it all together

In section 3.3.3, we have seen how its basic visibility and the content visibility in force at its location affect the effective visibility of an atomic block. To summarize it, we said that the basic visibility had priority, but in case it was

[4] Actually, the implementation of basic visibility descriptor and content visibility descriptor is not symmetric. Instead, there is a class representing content visibility descriptors which inherits from the basic visibility descriptor, thus aggregating the features of both in a single extended descriptor. However, this implementation detail does not alter in any way the semantics with respect to what we are explaining here.

undefined, the effective visibility would be determined by the current content visibility. The case where both of them are undefined should not happen unless the hint table is improperly designed.

The existence of visibility overriding meta-commands does of course change the game. Consider example 3.5:

Listing 3.5: *Eiffel*: Example of content visibility overriding

```
--# HIDE_NEXT_CONTENT if
if x > y then
  print ("Let me print something.")
  print ("Let me print something more.")
end
```

Clearly, what the teacher who wrote the annotation on the first line expects and wants, is the hiding the two 'print' instructions, as well as of the 'x > y' condition, which are atomic blocks. If it were true that the basic visibility of an atomic block, if defined, always takes priority over the applicable content visibility, this example would not produce the desired output. Thanks to AutoTeach's design, this is not the case.

As we know, when scanning the code, AutoTeach is always aware of what the current content visibility policy in the current location is. Every time an atomic block is encountered, AutoTeach does not need to check its parent block, it already knows everything it needs to know. Moreover, not only the current content visibility policy is stored in AutoTeach's status, but also its **specificity**, that is, whether the current policy comes from the content visibility table (default, unoverridden value), from a global override or from a local override.

As we saw, when an atomic block is encountered, we can easily look up its effective basic visibility and the *specificity of it* in the descriptor of the corresponding block type. And the final rule for determining its final effective visibility is:

For an atomic block, if its basic visibility and the content visibility policy at its location are both defined,* the one with the highest specificity *determines its final visibility. If both have the same specificity, then the basic visibility of the atomic block wins.

This rule is final. We finally know everything about the AutoTeach visibility model.

Figure 3.6 summarizes AutoTeach's visibility model, which we now finally see in its entirety. There are as many as six values that have a say on the final visibility of an atomic block, three of them related to its basic visibility and three related to the applicable content visibility policy at the location where the block starts[5].

[5]Once again, the content visibility in force at the starting location of the block is the same as the content visibility at the ending location, as both locations are contained in the same parent complex block.

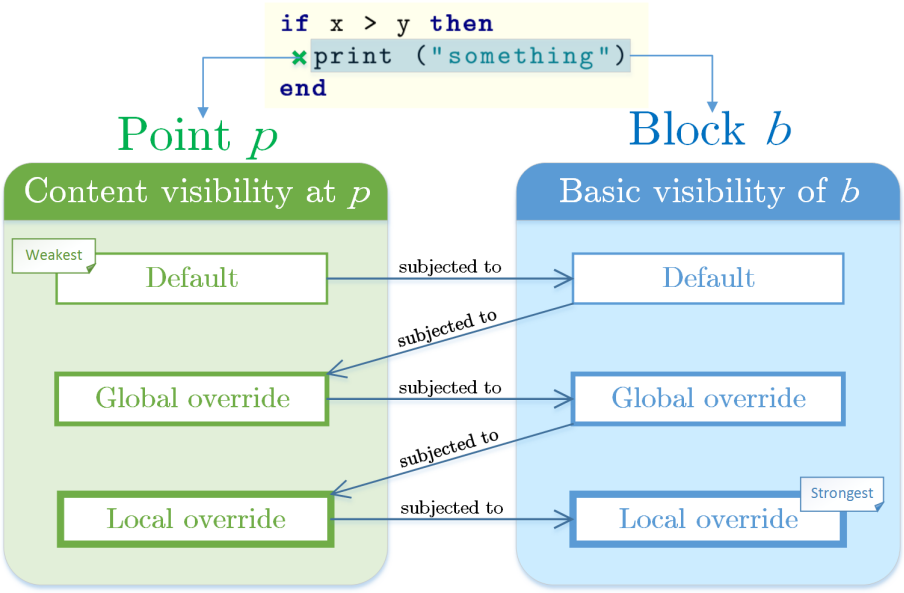

Figure 3.6: Determining the effective visibility of a block b located at point p.

Six factors playing a role might sound overwhelming, however the principle defining their priority is extremely intuitive (and, in our opinion, elegant): what is more specific is stronger. Clearly, a local override is more specific than a global override, which is in turn more specific than a default value. Also, basic visibility is in general a more specific concept than content visibility, as it defines the visibility of a single block, whereas content visibility defines the visibility of a whole region of code. For this reason, when values with the same specificity are compared, basic visibility is considered to be more specific.

3.7 Treating complex blocks as atomic

There is one last possibility offered by AutoTeach's processing which we have not seen yet and that will prove very useful in practice.

Imagine the following situation: you have an *if* block with a very trivial condition and body, something like this:

Listing 3.6: *Eiffel*: Example of a very simple *if* statement, which would be desirable to reveal all at once

```
if a > b then
    temp := a; a := b; b := temp
end
```

This can be expressed as "if 'a' is greater than 'b', swap them". Although this is technically a complex block, it is so simple that one might want to treat it as if it were a single instruction and make it appear all at once at some point

instead of having the keywords, the condition and the instructions revealed at different times. For what we know until now, this is only possible by adding several manual annotations.

Now imagine this second situation: we have an exercise where the students are provided with a class file and they are only asked to complete one (or some) feature of it, while all other features are provided to them as a part of the input. In such a case, we would like AutoTeach to print all the given features without any kind of processing and only process the feature(s) that the student should write herself. Issuing a "SHOW_ALL_CONTENT feature" command applied to the given features will not work, because we have seen that complex blocks for which the content visibility is explicitly defined will not inherit it from the parent blocks. In our example, even if we write "SHOW_NEXT_CONTENT feature" right before a routine, its body will still not be shown at lower hint levels, as the default table specifies a content visibility of False for blocks of type *routine_body*. We would need to issue a 'SHOW_NEXT_CONTENT' command for each complex block within the routine, which can be very annoying.

To address cases like these, we have included in AutoTeach the possibility of specifying that a particular a complex block (either a single occurrence or all complex blocks of a certain type) should be exceptionally treated as an atomic block. This is done with the following meta-commands:

```
--# TREAT_NEXT_AS_ATOMIC <complex_block_type>
--# TREAT_ALL_AS_ATOMIC <complex_block_type>
```

The reverse commands are:

```
--# TREAT_NEXT_AS_COMPLEX <complex_block_type>
--# TREAT_ALL_AS_COMPLEX <complex_block_type>
```

These are also only applicable to complex blocks.

Doing this will effectively transform the affected blocks into atomic blocks. The consequences of this transformation are exactly those that one would expect in conformity with what we have said until now. In particular:

- Atomic blocks can only be completely hidden or completely shown, there is no case in which an atomic block is partially shown. This also applies to complex blocks which are being treated as an atomic block, and is exactly what we wanted in the last two examples.
- Complex blocks are not subject to the content visibility in force at their location, but atomic blocks are. This means that, unlike regular complex blocks, an "atomicized" complex block can be made visible or hidden depending on the valid content visibility policy.

Let's get back to our examples and see how this extension helps. In the first case, a "TREAT_NEXT_AS_COMPLEX if" annotation will ensure two things:

- The if block will be shown all at once at some point, instead of being revealed incrementally.

- The if block will only be shown when the content visibility of the outer block allows it. Assuming that the if block is not nested within other complex instructions and that there are no overriding annotations, the if will appear when the content visibility of routine_body becomes True, which is when all the other non-nested instructions in the class become visible too.

Eventually, we achieved what we wanted: the whole *if* instruction will behave as if it were a single instruction

In the second case, we can simply add a global "TREAT_ALL_AS_ATOMIC feature" annotation at the beginning of the class, and then a single "TREAT _NEXT_AS_COMPLEX feature" annotation right before the feature(s) that students should complete. Since any visibility table will normally always show the existence of features, even on level zero, this will cause all features to be completely shown regardless of anything else - all but the one(s) that the student should implement.

The default behavior, unsurprisingly, is to treat all complex blocks as complex. This is intentionally not modifiable.

This was the last point to see. Our long tour through the AutoTeach features is now complete!

3.8 Final thoughts

Before moving to the next chapter, it is time to sum things up and make some considerations

3.8.1 Modularity of the model

The AutoTeach visibility model is clearly very powerful, but its complexity may sound overwhelming. In particular, the fact that as many as six flags play a role in determining the visibility of an atomic block can sound really scary! In practice however, in the great majority of cases, most of them will be left undefined and can safely be ignored. Even more importantly, the model can be simplified in several different ways, each of which leads to a perfectly consistent and simpler model. In particular:

- If you don't write any meta-commands, no override flag will ever be set, and only the visibility table will be taken into account. You can forget about the existence of override flags and stick to what has been presented in section 3.3. This will actually often be the case: if the table is smart enough, many exercises can be processed nicely without the need of touching anything. In our intentions, the default automatic hint table should be suitable for the majority of exercises.
- At the price of giving up some flexibility, the concept of content visibility can be completely ignored. It suffices to use a custom hint table where all

block content visibilities are undefined. If you do this, then you can forget about this concept, and only reason about basic visibility.

3.8.2 Content visibility inheritance

The power of content visibility inheritance is limited. Content visibility from an outer block will only be inherited if it is undefined in the inner block. If an inner block has a defined content visibility in the table, then there is nothing that can be done in order to have that block inheriting the content visibility from its parent. In fact, there is no command that will "undefine" it and/or force inheritance from the outer block in any other way. However, there are other ways to achieve the desired result, such as treating complex blocks as atomic.

3.8.3 Arbitrary choices

Several choices in the design of AutoTeach are discretionary. A notable example of this is the choice of blocks. Which blocks should be considered atomic and which ones complex? What are the exact boundaries of blocks? Should the parentheses surrounding argument declarations be considered a part of the *feature* or the of *arguments* block? For all those things, we just tried to choose a reasonable level of granularity and make the most sensible choices, bearing in mind what the ultimate goal of AutoTeach is: helping students. The future usage experience of AutoTeach can confirm the validity of some choices and lead to reconsider other choices: we have done our best not to hardcode these design choices too deep in the code and make it easy to change them in the future, if so wished.

Immunity of complex blocks to content visibility

A choice that could sound criticizable is making complex block immune to the content visibility of their parent block. The reason for this is that in our vision complex blocks make up the skeleton of the code, while atomic blocks make up the "flesh". When one wants to hide the content of something, she wants to hide its flesh, and in the case of inheritance this is propagated to inner blocks, but generally she does not want to hide the skeleton, not even small bones (nested complex block, in this metaphor). In those cases where one really wants to hide the skeleton, basic visibility is to be used.

There are some cases where it is immediately clear that this makes more sense: imagine how useless a "HIDE_NEXT_CONTENT feature" command would be if it caused the whole content of the feature besides its declaration to completely disappear! You probably prefer to hide all the leaves, but still leave the require, local, do, end, etc. keywords in place.

In other cases this might be less obvious, but we still believed that it made sense considering what the goal of AutoTeach is. If we have two nested loops, a "HIDE_NEXT_CONTENT loop" on the outer loop will not hide the existence

of the inner loop, the student will see the skeleton of the two nested loops, albeit with no contents. This is generally better than only showing the existence of the external loop, because if an exercise is to be solved with multiple nested loops, the student should see the real depth of the loops as soon as their structure is revealed. Showing only the outer loop might be misleading for the solution of the exercise.

More power than needed

Although you have full control on the visibility and content visibility of all types of blocks, in practice, there are several things that should never need to be touched. For example, there is probably no situation where one would want to hide (basic visibility) a *precondition* complex block, as this means that the 'require' keyword is also hidden. Even when a precondition is hidden, students should see the 'require' keyword, so that they know that a precondition is there, and the way to achieve this is to hide the *content* of the precondition. However, for consistency and completeness, AutoTeach allows the user to hide blocks of any type she wishes, including those that in practice should never be hidden.

In the next chapter we will explore some good practices of using AutoTeach that will help not to feel overwhelmed by its complexity.

Chapter 4

AutoTeach tips & tricks

In the previous chapter, we have extensively discussed all the principles on which AutoTeach is based. As a complement, this chapter will focus on practical use cases and good practices with AutoTeach, also making some didactic considerations.

4.1 Optimizing the hint table customization

As we know from section 3.4.1, a complete hint table should theoretically have as many as 41 rows. This would make it extremely hard to write custom hint tables. Fortunately, many of these rows never need to be touched in practice. The rest of this section (4.1) demonstrates this.

4.1.1 Visibility of atomic blocks

In general, *a well-built table should always leave the visibility of atomic blocks undefined and only work with the visibility (basic and content) of complex blocks.* This is a strong claim, which we will now motivate.

The first reason is that it is generally possible (and easy) to control the visibility of atomic blocks through the content visibility of their potential parents, and this alone would be a sufficient reason: since we do not lose anything by giving up the possibility of altering the visibility of atomic blocks, doing it will reduce the number of factors we have to take care of.

A second reason, as important as the first, is that, in some cases, not following this rule will preclude some possibilities, in particular in the case of *instruction* and *assertion* blocks.

The default automatic hint table has a level on which instructions appearing directly within a routine body are visible, however instructions contained in complex statements (such as *if*s and loops) are still hidden. This behavior can only be obtained by leaving the basic visibility of *instruction* undefined and ad-

justing the content visibility of blocks that can contain instructions (in particular *routine_body*, *if_branch*, *inspect_branch*, *loop_initialization* and *loop_body*). It is clear that this is the only way, as setting the basic visibility of *instruction* to *true* or *false* would cause them to be *always* shown or hidden. Since there are (or there should be) no situations where working with content visibilities alone cannot obtain something attainable by explicitly defining the basic visibility of *instruction*, this should be considered a general good practice and always be followed.

The case of assertions is very similar. Explicitly defining the visibility of assertions would make them visible or hidden regardless of the block in which they appear (*precondition*, *postcondition*, *loop_invariant*, *class_invariant*). For this reason, their visibility should always be left undefined, relying instead on the content visibility of blocks that can contain assertions. Of course, in some cases it might be desirable to make all assertions visible (or hidden) in any possible location where they can appear, however even in this case there is no advantage in doing this by acting on the basic visibility of assertions over relying on the content visibility of parent blocks. So, for consistency, the good practice of working with content visibility should not be broken.

For arguments and locals the story is slightly different. Blocks of type 'argument_declaration' can only occur inside complex blocks of type 'arguments'. Similarly, blocks of type 'local_declaration' can only occur inside a 'locals' block. In this case, working with the basic visibility of 'argument_declaration' or the content visibility of 'arguments' makes no practical difference. However, there is no drawback in working with content visibilities in this case, and doing this will allow us to stay consistent with the remaining cases, so we still recommend to work with content visibilities. A similar case is the one of blocks *loop_termination_expression* (which only appears inside *loop_termination*) and *loop_variant_expression* (only appearing inside *loop_variant*).

The only exception to this principle is block *if_condition*. This block can only appear inside an *if* complex block, however altering the *if*'s content visibility would also affect the other parts of the if. For this reason, it is generally better to specify the visibility of *if_condition* blocks directly.

Summing it up, we have six rows in the hint table which can be completely omitted if we want.

Clearly, what we have said in this section only applies to hint tables: altering the basic visibility of atomic blocks with annotations in the code is not considered a bad practice and is actually extremely useful.

4.1.2 Basic visibility of complex blocks

In the majority of the cases, the basic visibility of most complex blocks will be set to True for all levels and never be touched. This is another strong claim, which we will now motivate.

If you check out the complete list of complex blocks in the appendix (A.2), you will notice that the majority of complex blocks represent *a part of another complex block*. In fact, the only 'autonomous' complex blocks are *feature*, *if*,

loop and *inspect*. All the others represent a part of these four, for example *precondition* and *routine_body* are a part of *feature*, and *loop_initialization* and *loop_variant* are a part of *loop*.

Now, we believe that hiding one of such blocks would be deceiving for the students about the structure of the code. For example, completely hiding a precondition, including the 'require' keyword, would be unfair to students: if they are supposed to write a precondition, they should in the first place know that a precondition was there! Similarly, it is fine to completely hide a loop, but if the loop is shown and the loop has an invariant, we should not pretend that there is no invariant, we should always show its existence, possibly hiding its content.

Since the orphan rule (3.3.3) prevents any block from being shown if its parent is not visible as well, it is safe to set the basic visibility of all these blocks to *true* for all levels and forget about it.

Eventually, this argument applies to all complex blocks except *feature*, *if*, *loop* and *inspect*. However, there would almost always be no point in hiding a feature entirely, not at least by default, so we are left with *if*, *loop* and *inspect*. For all complex blocks except for these three, we should always set the basic visibility to *true* and forget about it. This means fourteen more table rows which we don't have to worry about.

Summing up the results of the last two sections, we almost halved the number of rows which we will really need to touch when defining a custom hint table. This, combined with the simplifications explained in section 3.4.1, greatly improves the manageability of custom hint tables.

For an example of the application of all these principles, please refer to the default automatic hint table source file (A.6).

4.2 Compacting hint levels

The default automatic hint table has as many as eight hint levels. In many cases, this can result in two consecutive levels producing identical outputs (for example, in an exercise where the code contains no contracts, the output of the a level that only adds visibility of contracts will be identical to the output of the previous level). In some other cases, even if this does not happen, it will still be desirable to reduce the number of hint levels.

Our recommendation for those cases where the default processing policy is satisfactory and it is only desired to reduce the number of hints is to take care of this by manually removing the unnecessary hint levels from the output. In the case of identical levels, this can be automatized very easily, and in fact as a part of this thesis we provide a simple tool, HintCompacter, which does exactly this.

HintCompacter is a simple command line tool which requires only two arguments: a directory and a minimum output hint level (an integer). When launched, it will scan the target directory looking for subfolders named after

natural numbers (which do not necessarily start from zero), containing the output of AutoTeach at different hint levels. These directories will be renamed with a new numbering starting from the specified minimum output hint level. If two or more levels have produced an exactly identical output, these levels will be merged together.

Even in cases where the process cannot be automatized, if the default policy is deemed satisfactory, it is much easier to manually compact the hint levels after running AutoTeach than defining a custom hint table just for this reason.

4.3 Other tips and tricks

4.3.1 Hybrid hints

As we saw in the introductory tour (section 2.3), textual hints can *and should* be integrated with other processing directives, so that they are well coordinated with code blocks entering the scene. AutoTeach offers this possibility, and it should be used whenever possible.

It is very important for students not only to read what they should do, but also to see with their eyes how they should do it. Saying "maybe you should use a loop" is not as effective as showing the bare skeleton of a loop, even if these two hints theoretically carry exactly the same information. However, it is also important to give the students a chance of translating a qualitative hint to real code. For this reason, when using textual hints, it is often a good idea to alternate between levels adding textual hints and levels revealing the part of the code that was suggested in the previous textual hint.

Do not be afraid of having too many different hints. If a hint is not useful to a student, she will simply request the next one straight away. On the other hand if a student receives too much additional information with a single hint, that will be probably spoil a part of the solution.

4.3.2 Sequences of instructions

By default, AutoTeach gives hints from the outside in. However, in some exercises, it might be necessary to incrementally reveal a *sequence* of instructions. There is no simple instruction for doing this. The recommended trick for doing this is to issue a single 'SHOW_ALL_INSTRUCTION' command at the beginning of the sequence and then several 'RESET_ALL_INSTRUCTION' commands in the sequence, with different validity level.

The following listing, taken from a real exercise, shows an example of application of the last two tips:

Listing 4.1: *Eiffel*: Example of a manually annotated exercise

```
--# SHOW_NEXT_CONTENT arguments
swap_attributes (other_book: BOOK)
```

```
    -- Swap title, author, and number of available copies of the
       current book with the corresponding attributes of
       other_book.
  --# [2] SHOW_NEXT_CONTENT locals
  local
    --# [1] HINT: When you swap the values of two variables,
    --#- you always need a temporary helper variable.
    --#- Here you have to swap three attributes,
    --#- so it's reasonable to use one variable for each of them.
    l_author, l_title: STRING
    l_number_of_available_copies: INTEGER
  do
    --# [3] HINT: Save the values of the attribute of this book
    --#-  to your temporary variables.
    --# [4] SHOW_ALL instruction
    l_author := author
    l_title := title
    l_number_of_available_copies := number_of_available_copies

    --# [4-5] RESET_ALL instruction
    --# [5] HINT: Now you can copy the values from 'other_book'
    --#- to this book.
    title := other_book.title
    author := other_book.author
    number_of_available_copies := other_book.
        number_of_available_copies

    --# [6-7] RESET_ALL instruction
    --# [7] HINT: Finally, you can set the attributes of
    --#- 'other_book' with the values you saved in the
    --#- temporary variables.
    other_book.set_title (l_title)
    other_book.set_author (l_author)
    other_book.set_number_of_available_copies (
        l_number_of_available_copies)
    --# [8-] RESET_ALL instruction
  end
```

Notice how the suggestion of 4.3.1 is applied here. On levels 1, 3, 5, and 7, a textual hint is made visible. On levels 2, 4, 6 and 8, the part of the code suggested by the previous hint is revealed. Also notice the usage of the 'SHOW_ALL' and 'RESET_ALL' commands.

4.3.3 Treating complex blocks as atomic

We saw in section 3.7 that AutoTeach can be instructed to treat complex blocks (either a single instance or all blocks of a certain type) as if they were atomic. This feature is a good candidate for being the least understood and used, while it can actually be very useful!

In section 3.7 we presented the most obvious use case, however there is at least another one which is very frequent. AutoTeach cannot be run on anything smaller than a class. However, many exercises only consist of completing a single feature within the class, while all the other features are known and given to the student.

Now, AutoTeach has no such command as 'SHOW_NEXT_CONTENT_FULL'

to force a block to be fully shown no matter what, because such a command would break the visibility model. However, there is an easy solution for instructing AutoTeach not to touch those features: having them processed as if they were atomic blocks. Since generally, the basic visibility of features is always *true*, adding a "`--# TREAT_NEXT_AS_ATOMIC feature`" annotation before every feature which should not be touched will be enough for doing the job, and doing it very easily! Even better, it is possible to add a single "`--# TREAT_ALL_AS_ATOMIC feature`" annotation at the beginning of the class and then a "`--# TREAT_NEXT_AS_COMPLEX feature`" annotation before the feature(s) that should be processed normally. All the other features will be printed to the output untouched.

Chapter 5

AutoTeach implementation

After having dealt with the use of AutoTeach both from the theoretical and the practical point of view, it is now time to have an insight at AutoTeach's internal implementation.

5.1 EVE

AutoTeach is implemented as a module for EVE (Eiffel Verification Environment, the research version of EiffelStudio [11]). The main reason for this choice was the necessity of accessing the great wealth of information generated by the Eiffel compiler. Making AutoTeach an independent tool would have created an external dependency on the compiler which would have been impractical to handle.

The main implication of this is that a full version of EVE is required for running AutoTeach and that all projects must be fully compiled before they can be processed by AutoTeach (whereas a solution based on an independent parser would only have required a successful parsing of the input code).

AutoTeach can be run by passing the proper arguments to the EVE executable. Please refer to the appendix for the command line documentation (A.1).

5.2 Classes

Figure 5.1 shows a simplified class diagram of AutoTeach, only containing the most important classes.

We will focus on two of them, 'AT_AST_ITERATOR' and 'AT_PROCESSING_ORACLE', and briefly explain what all the other do.

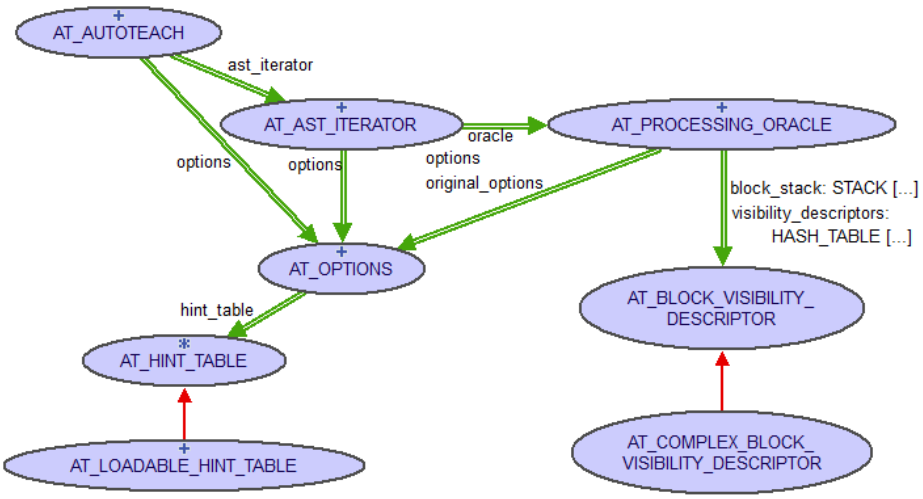

Figure 5.1: Simplified AutoTeach class diagram.

5.2.1 AT_AST_ITERATOR

As we know, AutoTeach processes classes sequentially, visiting their AST (abstract syntax tree). This class is where this processing really takes place.

Class 'AT_AST_ITERATOR' inherits from 'AST_ROUNDTRIP_PRINTER_VISITOR', a class in the Eiffel parser library. Class 'AST_ROUNDTRIP_PRINTER_VISITOR' can take an AST node (which can even be an entire class), visit every node of it and print the text of all terminal nodes to an internal text buffer. Class 'AT_AST_ITERATOR' inherits this behavior and redefines the visiting routines for the relevant nodes, so that the printing of certain regions can be suppressed and replaced by the insertion of a placeholder.

In the early phases of implementation, this class was also in charge of defining what parts of the code should be printed and what other should be hidden and replaced by a placeholder. As the complexity of the class kept growing, we realized that it was a better idea to give this task to another class ('AT_PROCESSING_ORACLE', see next) and relieve the present class from it. Every instance of 'AT_AST_ITERATOR' will create an instance of 'AT_PROCESSING_ORACLE', which will coherently with its name, acts as an oracle, i.e., it provides answers to questions. This oracle is kept up to date with all the relevant information about the processing status, and every time a block of a relevant type is encountered, the iterator informs the oracle that a block of a certain type is about to begin (or end). When the time comes of printing anything to the output, the iterator simply asks the oracle whether or not the text should be printed. In particular, the parent class has a '*put_string*' feature which is responsible for printing all terminal nodes by putting their text to the internal text buffer. Class 'AT_AST_ITERATOR' redefines it so that the oracle is consulted first.

Breaks (interruptions in the AST, generally whitespace or comments) are decomposed into single lines and searched for meta-commands. Every comment

starting with a hash (the symbol denoting meta-commands) is passed to the oracle for processing as a meta-command.

Class 'AT_AST_ITERATOR' also redefines a number of other features named '*process_*_as*', such as *process_feature_as* or *process_once_as*. The majority of these features simply notify the oracle that a block of a certain type is beginning, then make a precursor call and finally notify the oracle that the processing of that block is complete.

Class 'AT_AST_ITERATOR' also takes care of the correct printing of whitespace, new lines and the correct indentation of the output. While this task has proven to be complex and required a considerable effort, this topic is not particularly interesting for discussion here.

5.2.2 AT_PROCESSING_ORACLE

While class 'AT_AST_ITERATOR' was responsible with the handwork, class 'AT_PROCESSING_ORACLE' is responsible for most of the conceptual work. As we said, every instance of 'AT_AST_ITERATOR' needs an oracle to tell it at any time whether or not the current block should be printed. The oracle is given all the necessary information for answering this question. The oracle is also in charge of processing all meta-commands.

The oracle keeps track of the visibility and content visibility policy for the current region of code by using stacks. At the beginning of a new block, the oracle checks the hint table and the active overrides for that block type, combines it with the current visibility and the active content visibility policy, calculates the effective visibility of that block and pushes it to a stack. If the block is a complex block, the new content visibility valid within the block is also computed and pushed to another stack.

For keeping track of the active overrides, the oracle keeps a table of block visibility descriptors. Descriptors have been already explained in detail in sections 3.6.1 and 3.6.2. As anticipated in a footnote in 3.6.2, complex blocks don't have two separate descriptors for their basic and content visibility, instead they only have one descriptor of a more specific type, which has the relevant attributes for both basic and content visibility. As the active hint table might change during the processing, every descriptor is initialized with an agent (two in the case of complex blocks) for retrieving the default basic visibility (and content visibility in the case of complex blocks) from the current active table.

The oracle is also in charge of parsing meta-commands and act accordingly. Besides meta-commands acting on block visibility, this means that the oracle also takes care of switching to another hint table when a 'MODE' meta-command is encountered, and update the processing options if the insertion of the code placeholder is toggled. Textual hints are the only the type of meta-command which requires passing some information back to the iterator. This is done with a '*last_command_output*' string attribute in the oracle. This attribute is updated every time a meta-command is processed, and is normally set to *Void*, but if a hint is processed and the hint is applicable to the current level, this attribute is set to the text of the hint line that should be printed to the

output. After passing the command to the oracle, the AST iterator will print this line to the output.

5.2.3 Complete list of classes

- **General classes:**
  - **EWB_AUTOTEACH**: AutoTeach command line module, in charge of parsing the arguments and set options accordingly.
  - **AT_AUTOTEACH**: AutoTeach main class. Searchs for the classes to be processed, adds them to the processing list, creates the output file structure and feeds an instance of 'AT_AST_ITERATOR' with the AST of the classes to be processed.
  - **AT_AST_ITERATOR**: scans the AST (abstract syntax tree) of input classes, replacing some code blocks with placeholders. See section 5.2.1.
  - **AT_PROCESSING_ORACLE**: contains the processing logic for determining the visibility of code blocks, parses meta-commands. See section 5.2.2.
  - **AT_BLOCK_VISIBILITY_DESCRIPTOR** and **AT_COMPLEX_BLOCK_VISIBILITY_DESCRIPTOR**: descriptors used by the processing oracle for keeping track of the current visibility and content visibility overrides. See sections 5.2.2 and 3.6.2.
  - **AT_COMMAND**: class representing a meta-command. Only responsible for parsing the syntax and the validity level range, the payload and the semantics of commands are processed directly by the oracle.
  - **AT_OPTIONS**: contains the options for the execution of AutoTeach, such as the list of classes to be processed, the hint level and and the active hint table.
  - **AT_COMMON**: common ancestor to most AutoTeach classes, contains several utility functions and shared instances of objects (e.g. enumeration types).
  - **AT_STRINGS**: contains all string constants, including messages and localizable strings, used by AutoTeach.
- **Hint tables:**
  - **AT_HINT_TABLE**: deferred class representing a hint table.
  - **AT_LOADABLE_HINT_TABLE**: effective class implementing a hint table that can be loaded from a text file. There used to be two separate classes for the two default hint tables, but these are now also loaded from a default file through this class.
  - **AT_HINT_TABLES** and **AT_SHARED_HINT_TABLES**: helper classes providing shared instances of the default hint tables.
- **Utility classes:**

  - **AT_ENUM** and **AT_ENUM_VALUE**: deferred classes representing an enumeration type and an enumeration value. For more information about enumeration types, please proceed to the next section.
  - **AT_TRILEAN**: expanded class representing a trilean. For the semantic of trileans, please refer to section 3.1, for notes about the implementation of trileans please proceed to section 5.3.1.
  - **AT_TRILEAN_CONSTANTS**: class containing constants for the three valid trilean values (*true*, *false*, *undefined*).

- **Enumerations:** the following classes represent enumeration types.
  - **AT_ENUM_BLOCK_TYPE** and **AT_BLOCK_TYPE**: enumeration and value type for the block type enumeration. This enumeration contains all the supported block types, with extra queries for distinguishing complex blocks and atomic blocks.
  - **AT_ENUM_MODE** and **AT_MODE**: enumeration and value type for the mode enumeration. As a reminder, valid modes are 'auto', 'manual' and 'custom'.
  - **AT_ENUM_PLACEHOLDER** and **AT_PLACEHOLDER**: enumeration and value type for the placeholder enumeration. This enumeration contains the different placeholders to be inserted depending on what part of the code is omitted (in particular, *if* conditions and arguments require a special inline placeholder).
  - **AT_ENUM_POLICY_STRENGTH** and **AT_POLICY_STRENGTH**: enumeration and value type for the visibility policy strength enumeration. This enumeration contains the three different strength values for a visibility policy ('default', 'global override', 'local override'), which derive directly frrom the policy specificity.

5.3 Additional contributions to the Eiffel libraries

While working for this thesis we sometimes felt the necessity of extending the Eiffel framework with some classes that were needed for AutoTeach, but seemed applicable to a far wider context than AutoTeach alone. In this section we present an implementation of trileans and enumeration types for Eiffel, which we deem suitable for being integrated into other existing Eiffel libraries.

5.3.1 Trileans

We discussed the semantics of trileans in depth in section 3.1. Here we only mention some relevant information about their implementation.

The class representing a trilean value is 'AT_TRILEAN'. This is an expanded class with only two attributes: '*is_defined*' and '*internal_value*', both of them boolean. The '*is_defined*' attribute indicates whether the TRILEAN

is defined or not. The '*internal_value*' attribute contains the boolean value of the trilean and is only meaningful when '*is_defined*' is *true*.

As the name suggests, the '*internal_value*' is not exposed directly. Instead, it is accessed through the 'value' query, which, as a precondition, can only be called if '*is_defined*' is *true*.

Trileans supports direct conversion from boolean ('*convert*' clause). The reverse conversion is not supported, as not all trileans can be converted to booleans. When this is necessary, the '*value*' query should be called, after having checked that the trilean is actually defined.

Most operations defined on the boolean type have been defined for trileans as well. Unlike most ternary logic systems, such as Kleene's logic [15], the '*and*', '*or*' and '*xor*' operations have all been defined so that if one (and only one) of the two operands is undefined, the result of the operation is the other operand. However, none of these operators are used by AutoTeach, and can be modified if necessary if and when the class is integrated into another Eiffel library.

The subjection and imposition operations are also defined. Please refer to section 3.1.2.

Implication is not defined.

5.3.2 Enumeration types

Eiffel lacks support for enumeration types. There are several workarounds to this omission that work reasonably well in several situations, however we felt that in AutoTeach, particularly for the case of block types, none of these workarounds was satisfactory. For this reason, we implemented classes 'AT_ENUM' and 'AT_ENUM_VALUE', which aim to simulate enumeration types until proper support for them is implemented in the Eiffel language.

Class 'AT_ENUM [G -> AT_ENUM_VALUE]' represents an enumeration type, with values of type G. Class 'AT_ENUM_VALUE' represents a value for an enumeration.

In an enumeration, every value has a name (string) and a numerical value (integer). The enumeration type itself also has a name. The 'AT_ENUM' class provides features for easily retrieving the name of a value with a given numerical value or, symmetrically, getting the numerical value of a value of which we know the name. The list of all possible values can also be easily retrieved.

To declare an enumeration type, a new class inheriting from 'AT_ENUM' must be declared (e.g. 'AT_ENUM_MODE'). This class will have to redefine feature 'value_list', and implement a once routine which returns an array of tuples representing all the valid enumeration values, with their names and numerical values. Features finding a value given its name or numerical value must be redefined as well, however their implementation is trivial. The class takes care of the rest.

It is generally useful to declare a constant for every valid enumeration value in the enumeration class, for quick access from within the code.

It is also necessary to declare a subclass of class 'AT_ENUM_VALUE'. The subclass should be an expanded type, and should be named conformingly to the entity which it should represent (e.g. 'AT_MODE'). The only feature that should be redefined is '*enum_type*', which should create an instance of the corresponding enumeration type (in our example, 'AT_ENUM_MODE'). The two initialization procedures as well as the 'default_create' procedure should be listed in the 'create' clause.

Contracts declared in class 'AT_ENUM_VALUE' will ensure that enumeration values are consistent with the valid values declared by the enumeration type.

While this approach requires certainly more work than what real, built-in enumeration support for Eiffel would require, it has an important advantage: enumeration classes can be freely customized. This makes it possible, for example, to define the distinction between atomic and complex blocks directly within the enumeration class.

Listing 5.1 shows an example of an enumeration type class.

Listing 5.1: *Eiffel*: Example of an enumeration type class

```
class
  AT_ENUM_PLACEHOLDER

inherit

  AT_ENUM [AT_PLACEHOLDER]
    redefine
      name
    end

feature -- Access

  name: STRING = "placeholder"
      -- <Precursor>

  value (a_value_name: STRING): AT_PLACEHOLDER
      -- The value with name 'a_value_name'.
    do
      create Result.make_with_value_name (a_value_name)
    end

  value_from_number (a_numerical_value: INTEGER): AT_PLACEHOLDER
      -- The value with numerical value 'a_numerical_value'.
    do
      create Result.make_with_numerical_value (a_numerical_value)
    end

feature {AT_ENUM} -- Value list

  value_list: ARRAY [TUPLE [numerical_value: INTEGER; value_name:
      STRING]]
      -- Effective list of values.
    once ("PROCESS")
      Result := <<  [0, "no_placeholder"],
                [1, "standard_placeholder"],
                [2, "arguments_placeholder"],
                [3, "if_condition_placeholder"]   >>
    end
```

```
feature -- Values

  Ph_none: AT_PLACEHOLDER
    once ("PROCESS")
      create Result.make_with_numerical_value (0)
    end

  Ph_standard: AT_PLACEHOLDER
    once ("PROCESS")
      create Result.make_with_numerical_value (1)
    end

  Ph_arguments: AT_PLACEHOLDER
    once ("PROCESS")
      create Result.make_with_numerical_value (2)
    end

  Ph_if_condition: AT_PLACEHOLDER
    once ("PROCESS")
      create Result.make_with_numerical_value (3)
    end

feature -- Properties

  placeholder_text (a_placeholder: AT_PLACEHOLDER): STRING
      -- Text of `a_placeholder'.
    do
      -- Implementation omitted.
    end

    is_inline (a_placeholder: AT_PLACEHOLDER): BOOLEAN
      -- Is `a_placeholder' an inline placeholder?
    do
      Result := inline_placeholders.has (a_placeholder)
    end

feature {NONE} -- Implementation

  inline_placeholders: ARRAY [AT_PLACEHOLDER]
      -- List of placeholders that must be inserted inline.
    once ("PROCESS")
      Result := <<Ph_none, Ph_if_condition>>
    end

end
```

The class shown in this listing is used in AutoTeach for representing the different supported types of code placeholder.

As we can see, the following features, which are declared as *deferred* by the parent class, are be redefined:

- '***name***': defines a name for the enumeration.
- '***value***' and '***value_from_number***': these two functions return an enumeration value (of type 'AT_PLACEHOLDER') given its string name or numerical value. Their implementation is trivial, however they cannot not be easily implemented by the parent class, as the parent class does not know the effective type of enumeration values that should be instantiated.

- '***value_list***': defines the list of all the valid enumeration values, in the form of tuples.

The remaining features of the class are optional. Feature under clause '*Values*' are a shortcut for getting each of the valid enumeration values directly, and make it possible to access them by writing, for example, "`enum_placeholder.Ph_standard`", which is similar to the syntax used in other languages for enumerations.

Finally, features '*placeholder_text*' and '*is_inline*' show how it is possible to implement custom properties for enumeration values, something that many programming languages do not allow.

Chapter 6

Conclusions

6.1 Conclusions

In this thesis we have presented an incremental hint system for Eiffel which is ready for being used in the 2014 edition of the Introduction to Programming course at the ETH Computer Science department. The exercises processed during the development of this thesis produced very satisfactory results, and we believe that a very good balance between effective automatic processing and flexible manual tuning has been achieved. The effectiveness of these kind of incremental hints on students will be assessed during the upcoming semester.

Besides the implementation of AutoTeach for Eiffel, another significant contribution of this thesis is the definition of a theoretical model for processing the source code, decomposing it in blocks and determining which of them should be visibile in the output. This model is powerful and general, with no particular obstacles preventing it from being applied to other programming languages in the future.

6.2 Future Work

AutoTeach is a command line tool and includes no GUI. Clearly, more work is needed to create an effective user interface for students, to be integrated in the Introduction to Programming MOOC, enabling students to request hints for the exercise they are trying to solve.

In doing this, it will be very important to record as much usage and statistical data as possible, as this is the only way to assess the effectiveness of hints. In particular, it is relevant to know:

- For every exercise, what fraction of students request at least one hint?
- How many hints do students request on average for a single exercise? (excluding those who do not request any)

- After receiving a hint, how long do students try to solve the exercise before requesting another one?

Analyzing these usage data will also help answer more specific questions about single exercises. For example, if the data show that most students tend to ask hints up to level 4 for an exercise, and then most of them solve it correctly, probably something is being revealed at level 4 which is particularly hard for students to guess. Such situations are not always obvious when designing exercises, and these data will greatly help detecting them.

Besides the user interface for students, a useful addition for AutoTeach could be a GUI for teachers. This GUI could show a live preview of the results of processing an exercise at all hint levels at the same time and would greatly facilitate the teachers' work.

Finally, the heritage of the AutoTeach block and visibility model is something that could be taken up for implementation for other programming languages. All the concepts are general and would need little or no adaptation for being applied to other languages commonly used in programming courses, as Java or C.

There is ample room for additional development of effective static code analysis tools usable for didactic purposes. A first step in this directions was Stefan Zurfluh's Eiffel Inspector [17], which is however a general-purpose code analyzer. Our contribution was limited to making some changes to it, by simplifying batch processing of code with single rules (or small subsets of rules). This makes it easier to implement rules for checking patterns which are only applicable to specific exercises (e.g. a rule enforcing that some code is recursive). We also contributed eleven general code analysis rules, several of which are particularly suitable for programming students or programmers coming to Eiffel from other languages.

In the future, Eiffel Inspector could be forked to a custom code analysis tool, which should be able to process students' code by loading on the fly the list of rules applicable to that exercise and new rules, strictly focused on didactics, could be added.

6.3 Related Work

The idea of incremental hints for programming exercises seems to be relatively new, and there doesn't seem to be any considerable academic literature about this topic. Even though the concept is not completely novel, nobody has discussed this topic from a theoretical and academical point of view as we did in this thesis, at least not so directly. This was one of the motivating factors for this work.

On the other hand, the topic of automated assessment of student solutions to programming exercises, has seen much more interest by the research community. Although focused on more specific situations, papers dating as back as 1969 [4] can be found.

A good starting point is the 2010 paper by Ihantola et. al. [5], with a good review of the situation of the automated assessment tools in the preceding four years and the literature about them. An interesting finding by this paper is that the fragmentation of these systems is negatively affecting their development: considerably better results could be obtained by joining efforts and open existing systems up for extensibility.

In [6], a system is presented for assessing the correctness of student solutions to introductory programming exercises. This is done by devising an error model based on potential corrections to errors that students might make. The score of the students is tightly related to the minimal set of corrections that must be applied to the submitted code to reach the correct solution.

A very interesting paper is [13]. In this paper, an automated grading system is presented which combines the results of three different approaches: unit testing, software verification and control flow similarity measurement (against a reference solution). The results of the study show a high correlation between automatic grades and grades assigned manually by teachers to the same code.

It is worth noting that both the two preceding papers take a different approach from the rule-based static analysis which we discussed in this thesis.

One last related topic worth mentioning is automated plagiarism detection in programming exercises. It is generally useful to detect students copying or sharing solutions with each other in the same course, however for some well-known problems copy-pasting from the internet can be an issue. In the introduction, we already mentioned MOSS (Measure Of Software Similarity, [2]), a well-known tool among teachers which has been around for about 20 years at the time of writing. Papers [1] have also been published on this topic.

Appendix A

AutoTeach reference

A.1 Command line arguments and syntax

AutoTeach is a part of EVE (Eiffel Verification Environment, practically the research version of EiffelStudio). AutoTeach can be run by executing the EVE executable (normally called 'ec'/'ec.exe') and providing the `-auto-teach` switch. If this is done, all the subsequent arguments are passed to AutoTeach.

The input of AutoTeach is a complete project. The output is, for every hint level, a file for every processed class, with the same name of the respective input class file.

Before AutoTeach can run, the project must be compiled. The appropriate arguments must be supplied to EVE so that the correct project is loaded and compiled. We list here the EVE command line arguments and switches which are most frequently used with AutoTeach. For an exhaustive list, please refer to the EiffelStudio documentation [9].

- "`-config PATH_TO_ECF_FILE`": specifies the path to the input project (ecf) file.
- "`-project_path OUTPUT_PATH`": specifies the output directory for the compiled project (which is generally **not** the output directory for the output produced by AutoTeach).
- "`-clean`": specifies that a clean compile is to be performed.

Following all EVE's standard arguments, the `-auto-teach` must be specified, followed by the switches and arguments listed here:

- "`-at-class CLASS_LIST`" or "`-at-classes CLASS\_LIST`": **mandatory**. Specifies the list of classes to be processed. If more than one class is provided, the list must be wrapped in double quotes and class names must be separated by a space. If only one class is specified, double quotes are optional.

- "`-at-hint-level HINT_LEVEL_RANGE`": optional. Specifies the hint level(s) on which AutoTeach should be run. 'HINT_LEVEL_RANGE' can either be a single natural number (zero included) or a range, indicated by two dash-separated natural numbers. The default hint level is zero.
- "`-at-output-path PATH`": optional. Specifies the directory where AutoTeach will generate the output files. The default is the current directory.
- "`-at-level-subfolders PATH`": optional. Requests AutoTeach to create a folder for each hint level it is run on, and place the output of each level in the respective folder. The folders are created as subfolders of the output directory, and are simply called '0', '1', etc.

  If this switch is omitted and a range of two or more hint levels has been specified, AutoTeach will prepend the hint level followed by an underscore to the name of every generated file, as otherwise all files would have the same name for different hint levels. A notice is printed to inform the user about this.
- "`-at-mode MODE`": optional. Specifies the execution mode for AutoTeach. Valid values for 'MODE' are 'auto', 'manual' and 'custom'. The execution mode can be temporarily overridden with meta-commands within a single class. Default is 'auto'.
- "`-at-custom-hint-table PATH_TO_TABLE`": optional. Specifies the path to a custom hint table file to be loaded. Only if a custom table is loaded will it be possible to run AutoTeach in custom mode. No custom table will be loaded if this switch is not provided.

The following example shows the full command line of an execution of AutoTeach:

```
ec.exe -clean -config "E:\AtInput\converter\converter.ecf" -
    auto-teach -at-output-path "E:\AtOutput\converter" -at-
    class DECIMAL_TO_BINARY_CONVERTER -at-hint-level 0-10 -at
    -level-subfolders -at-custom-hint-table "E:\AtInput\
    converter\custom_table.txt" -at-mode custom
```

A.2 Complete list of blocks

The following list contains all block types supported by AutoTeach, named by their AutoTeach name. Atomic blocks are in ***bold***.

- *feature*: a whole Eiffel feature. Starts with the feature name, includes the type declaration if any, extends up to the 'end' keyword if any.
- *arguments*: the whole argument declaration region of a routine, spanning from the open to the closed parenthesis symbol.
- ***argument_declaration***: atomic block. A declaration of one or more arguments of the same type (such as `a, b: INTEGER`).
- *precondition*: the whole body of a precondition of a routine, including the 'require' keyword.
- *locals*: the whole locals declaration region of a routine, including the 'local' keyword.
- ***local_declaration***: atomic block. A declaration of one or more locals of the same type (such as `a, b: INTEGER`).
- *routine_body*: the body of a routine, including the 'do', 'once' or 'deferred' keyword. The 'end' keyword is not a part of this block.
- *postcondition*: the whole body of a postcondition of a routine, including the 'ensure' keyword. The 'end' keyword is not a part of this block.
- *class_invariant*: a class invariant, including the initial 'invariant' keyword. The final 'end' keyword is considered to be a part of the class, not of the invariant, and therefore not of this block.
- ***assertion***: atomic block. An assertion (including its tag) in a pre/post-condition or in a loop/class invariant.
- ***instruction***: atomic block. An instruction of any type with the exception of *if* and *inspect* instruction and loops.
- *if*: an *if* instruction, starting with the 'if' keyword and ending with the 'end' keyword.
- ***if_condition***: atomic block. The boolean condition of an *if* instruction.
- *if_branch*: the *then* or *else* branch of an *if* instruction. Includes the 'then' or the 'else' keyword. Does not include the final 'end' keyword.
- *inspect*: an *inspect* statement, starting with the 'inspect' keyword and ending with the 'end' keyword.
- *inspect_branch*: a branch of an *inspect* statement, starting with the 'when' or 'else' keywoard and including the whole body of the branch (but not the final 'end' keyword).

- *loop*: a loop, starting with the 'from' or 'across' keyword and ending with the final 'end' keyword.
- *loop_initialization*: the initialization part of a loop, starting with the 'from' keyword and including the whole body.
- *loop_invariant*: a loop invariant, starting with the 'invariant' keyword and including the whole body.
- *loop_termination*: the 'until' part of a loop, including the 'until' keyword and the boolean condition expression.
- ***loop_termination_expression***: atomic block. The boolean termination condition of a loop.
- *loop_body*: the body of a loop, starting with the 'loop' keyword and including the whole body (but not the final 'end' keyword.
- *loop_variant* a loop variant, including the 'variant' keyword and the integer variant expression.
- ***loop_variant_expression*** atomic block. The integer loop variant expression.

A.3 Meta-command syntax

We provide here the full BNF syntax specification of meta-commands. For a friendlier explanation please refer to section 3.5. For a compact and complete list of all the supported meta-commands, please refer to section 3.5.3 of this appendix.

⟨*meta_command*⟩ ::= '`--`' ⟨*ow*⟩ '`#`' ⟨*ow*⟩ [⟨*level_range*⟩] ⟨*command_body*⟩ ⟨*EOL*⟩

⟨*level_range*⟩ ::= '`[`' ⟨*level_range_body*⟩ '`]`'

⟨*level_range_body*⟩ ::= ⟨*single_level*⟩
| ⟨*level_interval*⟩

⟨*single_level*⟩ ::= ⟨*natural*⟩

⟨*level_interval*⟩ ::= [⟨*natural*⟩] '`-`' [⟨*natural*⟩]

⟨*command_body*⟩ ::= ⟨*command_with_block*⟩
| ⟨*command_with_complex_block*⟩
| ⟨*hint_command*⟩
| ⟨*hint_continuation_command*⟩
| ⟨*meta_comment_command*⟩
| ⟨*placeholder_command*⟩
| ⟨*mode_command*⟩

⟨*command_with_block*⟩ ::= ⟨*command_with_block_keyword*⟩ ['`:`'] ⟨*whitespace*⟩ ⟨*block_type_list*⟩ ⟨*ow*⟩

⟨*command_with_block_keyword*⟩ ::= '`SHOW_ALL`'
| '`HIDE_ALL`'
| '`RESET_ALL`'
| '`SHOW_NEXT`'
| '`HIDE_NEXT`'

⟨*block_type_list*⟩ ::= ⟨*block_type_keyword*⟩ (⟨*whitespace_with_comma*⟩ ⟨*block_type_keyword*⟩)* [⟨*whitespace_with_comma*⟩]

⟨*block_type_keyword*⟩ ::= '`feature`'
| '`arguments`'
| '`argument_declaration`'
| '`precondition`'
| '`locals`'
| '`local_declaration`'

| 'routine_body'
| 'postcondition'
| 'class_invariant'
| 'assertion'
| 'instruction'
| 'if'
| 'if_condition'
| 'if_branch'
| 'inspect'
| 'inspect_branch'
| 'loop'
| 'loop_initialization'
| 'loop_invariant'
| 'loop_termination'
| 'loop_termination_expression'
| 'loop_body'
| 'loop_variant'
| 'loop_variant_expression'

⟨*command_with_complex_block*⟩ ::= ⟨*command_with_block_keyword*⟩ [':']
⟨*whitespace*⟩ ⟨*complex_block_type_list*⟩ ⟨*ow*⟩

⟨*command_with_complex_block_keyword*⟩ ::= 'SHOW_ALL_CONTENT'
| 'HIDE_ALL_CONTENT'
| 'RESET_ALL_CONTENT'
| 'SHOW_NEXT_CONTENT'
| 'HIDE_NEXT_CONTENT'
| 'TREAT_ALL_AS_ATOMIC'
| 'TREAT_ALL_AS_COMPLEX'
| 'TREAT_NEXT_AS_ATOMIC'
| 'TREAT_NEXT_AS_COMPLEX'

⟨*complex_block_type_list*⟩ ::= ⟨*complex_block_type_keyword*⟩
(⟨*whitespace_with_comma*⟩ ⟨*complex_block_type_keyword*⟩)*
[⟨*whitespace_with_comma*⟩]

⟨*complex_block_type_keyword*⟩ ::= 'feature'
| 'arguments'
| 'precondition'
| 'locals'
| 'routine_body'
| 'postcondition'
| 'class_invariant'
| 'if'
| 'if_branch'
| 'inspect'
| 'inspect_branch'
| 'loop'

| ‘`loop_initialization`’
| ‘`loop_invariant`’
| ‘`loop_termination`’
| ‘`loop_body`’
| ‘`loop_variant`’

⟨*hint_command*⟩ ::= ‘`HINT`’ [‘`:`’] ⟨*whitespace*⟩ ⟨*free_text*⟩

⟨*hint_continuation_command*⟩ ::= ‘`-`’ [‘`:`’] ⟨*whitespace*⟩ ⟨*free_text*⟩

⟨*meta_comment_command*⟩ ::= ‘`#`’ [‘`:`’] ⟨*whitespace*⟩ ⟨*free_text*⟩

⟨*placeholder_command*⟩ ::= ‘`PLACEHOLDER`’ [‘`:`’] ⟨*whitespace*⟩ ⟨*boolean_string*⟩ ⟨*ow*⟩

⟨*boolean_string*⟩ ::= ‘`true`’
| ‘`false`’
| ‘`on`’
| ‘`off`’

⟨*mode_command*⟩ ::= ‘`MODE`’ [‘`:`’] ⟨*whitespace*⟩ ⟨*mode_string*⟩ ⟨*ow*⟩

⟨*mode_string*⟩ ::= ‘`auto`’
| ‘`manual`’
| ‘`custom`’

⟨*whitespace*⟩ ::= ⟨*white_char*⟩ ⟨*whitespace*⟩ | ⟨*white_char*⟩

⟨*ow*⟩ ::= ‘’ | ⟨*whitespace*⟩

⟨*white_char*⟩ ::= ‘ ’ | ⟨*tab*⟩

⟨*whitespace_with_comma*⟩ ::= ⟨*white_or_comma*⟩ ⟨*whitespace_with_comma*⟩ | ⟨*white_or_comma*⟩

⟨*white_or_comma*⟩ ::= ⟨*white_char*⟩ | ‘`,`’

⟨*tab*⟩ ::= ? tab character ?

⟨*natural*⟩ ::= ? string parsable to a natural number ?

⟨*free_text*⟩ ::= ? any string not containing EOL characters ?

⟨*EOL*⟩ ::= ? end of line ?

A.4 Meta-command reference

A.4.1 Syntax

Basic syntax:

```
--# [x-y] COMMAND_KEYWORD <payload>
```

The '`[x-y]`' part is optional and indicates the minimum and maximum hint level for the command to be valid. If omitted, the command is valid at any hint level. Examples:

- `[2-5]`: from level 2 to level 5.
- `[4-]`: from level 4 to infinity.
- `[-4]`: up to level 4 (from level zero to level 4).
- `[4]`: from level 4 to infinity (equivalent to `[4-]`).
- `[-]`: from level 0 to infinity (equivalent to omitting the level range).

A hint level range can be specified for all commands. For simplicity, it will be always be omitted in the listings in this appendix.

For all commands requiring a block type as payload, `<block_type>` means "any valid AutoTeach block type identifier", as listed in A.2. `<complex_block_type>` means that the identifier of a complex block type is expected.

Also, for all these commands, a compact form is supported which will apply the same command to multiple block types. It is sufficient to provide multiple block types on a single line, separated by commas and/or whitespace, like this:

```
--# SHOW_NEXT instruction, if, if_condition
```

This will be automatically expanded to:

```
--# SHOW_NEXT instruction
--# SHOW_NEXT if
--# SHOW_NEXT if_condition
```

A.4.2 Visibility

Global visibility override

```
--# SHOW_ALL <block_type>
--# HIDE_ALL <block_type>
--# RESET_ALL <block_type>
```

Set the global visibility override for the specified block type respectively to *true*, *false* and *undefined*.

Global content visibility override

```
--# SHOW_ALL_CONTENT <complex_block_type>
--# HIDE_ALL_CONTENT <complex_block_type>
--# RESET_ALL_CONTENT <complex_block_type>
```

Set the global content visibility override for the specified complex block type respectively to *true*, *false* and *undefined*.

Local visibility override

```
--# SHOW_NEXT <block_type>
--# HIDE_NEXT <block_type>
```

Set the local visibility override for the next instance of the specified block type respectively to *true* and *false*.

Local content visibility override

```
--# SHOW_NEXT_CONTENT <complex_block_type>
--# HIDE_NEXT_CONTENT <complex_block_type>
```

Set the local content visibility override for the next instance of the specified complex block type respectively to *true* and *false*.

A.4.3 Treating complex blocks as atomic

```
--# TREAT_ALL_AS_COMPLEX <complex_block_type>
--# TREAT_ALL_AS_ATOMIC <complex_block_type>
```

From now on, treat all occurrences of blocks of type ‘¡complex_block_type¿’ respectively as complex blocks (default behavior) or as atomic blocks.

```
--# TREAT_NEXT_AS_COMPLEX <complex_block_type>
--# TREAT_NEXT_AS_ATOMIC <complex_block_type>
```

Regardless of the current processing policy, treat the single next occurrence of a block of type ‘¡complex_block_type¿’ respectively as a complex block or as an atomic blocks.

A.4.4 Other commands

MODE

```
--# MODE newmode
```

Switch to the specified mode until the end of the current class. Valid values for 'newmode' are 'auto', 'manual' and 'custom'.

PLACEHOLDER

```
--# PLACEHOLDER (on | off | True | False)
```

Enable or disable the insertion of a code placeholder when a region of code is skipped. Note that a blank line is always inserted regardless of placeholders being enabled or not.

HINT and hint continuation

```
--# [4] HINT: First line of hint text.
--#-  Second line of hint text.
```

Print a textual hint to the output. If a hint spans more than one line, lines following the first one must start with '`--#-`'. The hint continuation construct is only allowed if the preceding meta-command in the source code is not a hint.

Meta-comments

```
--## Meta-comment text.
```

This command does nothing. However, unlike normal Eiffel comments, it will not appear in the output file.

A.5 Custom hint table file format

Custom hint tables can be loaded from a plain text file formatted how explained here.

Empty lines or lines only containing whitespace characters are ignored. Lines starting with a hash ('#') symbol are considered comments and are also ignored.

On every non-ignored line, elements are separated by whitespace (one or more space or tab characters). The parsing is completely case-insensitive.

Every non-ignored line starting with word 'content' represents a row in the content visibility table. All other lines represent a row in the basic visibility table. Except for the additional leading 'content' word, the syntax of the two types of row is the same.

Remember that a row in a visibility table corresponds to a block type (only complex block types are allowed in the content visibility table). Columns correspond to hint levels. The number of columns is not fixed, that is, some rows can be longer than others. Attempts to access a row outside of its upper bound will return the value of the last defined cell of the row.

Following the optional 'content' word, the name of the block type is expected. In case of 'content' rows, this must be a complex block type. Following the block type name, a list of one or more trilean values, separated by whitespace, is expected. The row is zero-based, meaning that the first of these values will go into the zeroth cell.

Valid string representation of trilean values are:

- For *true*: 'true', 't', 'yes', 'y'
- For *false*: 'false', 'f', 'no', 'n'
- For *undefined*: 'undefined', 'u', '?'

The parsing of trileans is also case-insensitive.

Example of the syntax:

```
# Hint level:             0    1    2
            feature       T
content     feature       U

            arguments     T
content     arguments     F    T

            precondition  T
content     precondition  F    F    T
```

A.6 Default hint table for automatic mode

Full contents of file ***default_auto_table.txt***. The syntax is the same explained in A.5.

The behavior of the default hint table for automatic mode is explained in section 3.4.3.

```
# This is the default table used in automatic mode, that is, when the users trusts AutoTeach
# to do the job. It implements a (hopefully) reasonable default, general-purpose policy,
# revealing the code gradually. The user can still override whatever he wishes with annotations.
#
# DO NOT EDIT THIS FILE UNLESS YOU KNOW WHAT YOU ARE DOING.
# This file should only be edited by a developer who is modifying AutoTeach and wants to
# change its default policies. Users of AutoTeach willing to use a different policy from
# AutoTeach's default should load a custom table instead of modifying this file.
#
#
#
# Summary
# -----------
# Level 0: only show the skeleton of features
# Level 1: show all routine arguments.
# Level 2: show all contracts
# Level 3: show all local declarations
# Level 4: show the existence of compound statements (`if', `inspect', loops)
# Level 5: show the secondary clauses of all compound statements (`if_condition',
#          `loop_initialization', `loop_invariant', `loop_termination_expression', etc.)
# Level 6: show all instructions in routines which are outside compound statements.
#          Keep hiding instructions within `if_branch', `loop_body', `inspect_branch'.
# Level 7: show everything
#
# Levels in the above summary are incremental. Whatever is shown at level 'x', is implicitly intended
```

```
# to be shown at level 'x' + 1 as well.
#


# ------------------------- Complex blocks -------------------------

# Hint level:                       0       1       2       3       4       5       6       7
            feature                 T
content     feature                 U

            arguments               T
content     arguments               F       T

            precondition            T
content     precondition            F       F       T

            locals                  T
content     locals                  F       F       F       T

            routine_body            T
content     routine_body            F       F       F       F       F       F       T

            if                      F       F       F       F       T
content     if                      U
            if_condition            F       F       F       F       F       T
            if_branch               T
content     if_branch               F       F       F       F       F       F       F       T

            inspect                 F       F       F       F       T
content     inspect                 U
            inspect_branch          T
```

```
content     inspect_branch              F       F       F       F       F       F       F       T

            loop                        F       F       F       F       T
content     loop                        U
            loop_initialization         T
content     loop_initialization         F       F       F       F       F       T
            loop_termination            T
content     loop_termination            F       F       F       F       F       T
#           See below for loop_termination_expression
            loop_invariant              T
content     loop_invariant              F       F       F       F       F       T
            loop_body                   T
content     loop_body                   F       F       F       F       F       F       F       T
            loop_variant                T
content     loop_variant                F       F       F       F       F       T
#           See below for loop_variant_expression

            postcondition               T
content     postcondition               F       F       T

            class_invariant             T
content     class_invariant             F       F       T



# ------------------------- Atomic blocks -------------------------
# The visibility of atomic blocks should in general not be fixed.
# Instead, it should be determined by the content visibility of
# the location where they appear.
```

```
# Hint level:                          0
            argument_declaration       U
            local_declaration          U
            assertion                  U
            instruction                U
            loop_termination_expression U
            loop_variant_expression    U
```

A.7 Default hint table for manual mode

Full contents of file ***default_manual_table.txt***. The syntax is the same explained in A.5.

The behavior of the default hint table for automatic mode is explained in section 3.4.3.

```
# This is the default table used in manual mode. Manual mode means that everything
# is handled manually by the user through annotations.
# This table shows the bare skeleton of features (including arguments) and hides all the rest,
# giving total freedom to the user to do what he wants with manual annotations (and possibly textual
   hints).
#
# DO NOT EDIT THIS FILE UNLESS YOU KNOW WHAT YOU ARE DOING.
# This file should only be edited by a developer who is modifying AutoTeach and wants to
# change its default policies. Users of AutoTeach willing to use a different policy from
# AutoTeach's default should load a custom table instead of modifying this file.
#
#
#
# Summary
# -----------
# Level 0: show the skeleton of features, including all arguments. Hide all the rest.
#


# ------------------------- Complex blocks -------------------------

# Hint level:                           0

# Always show features...
            feature                     T
```

```
# ...but always hide their content...
content     feature                     F

# ...with the exception of arguments:
            arguments                   T
content     arguments                   T

# The following complex blocks should be visible, but without their content:
            precondition                T
            locals                      T
            routine_body                T
            postcondition               T
            class_invariant             T

# If, inspect and loop blocks are hidden, but their inner complex blocks
# (e.g. branches) must be shown if the parent blocks are shown:
            if                          F
            if_branch                   T

            inspect                     F
            inspect_branch              T

            loop                        F
            loop_initialization         T
            loop_invariant              T
            loop_termination            T
#           See below for loop_termination_expression
            loop_body                   T
            loop_variant                T
#           See below for loop_variant_expression
```

```
# ------------------------- Atomic blocks -------------------------
# The visibility of atomic blocks should in general not be fixed.
# Instead, it should be determined by the content visibility of
# the location where they appear.

# Hint level:                          0
            argument_declaration       U
            local_declaration          U
            assertion                  U
            instruction                U
            loop_termination_expression U
            loop_variant_expression    U
```

Appendix B

Eiffel Inspector: the new rules

In this appendix we present the 11 new rules that we have implemented for the Eiffel Inspector code analysis tool as a part of the work of this thesis.

B.1 Eiffel Inspector

Eiffel Inspector ([17], formerly known as Inspector Eiffel) is a static code analysis tool for Eiffel. It is implemented as a part of EiffelStudio, the Eiffel integrated development environment [8]. The tool is integrated with the Eiffel compiler, which makes it possible for it to access all the internal semantic information generated by the compiler.

Eiffel Inspector initially came with a first set of code analysis rules. It is however easily extensible, and several other rules have been implemented in the months following its release.

Eiffel Inspector is a general purpose code analysis tool, however nothing forbids to create rules targeting specific aspects that are only applicable in particular situations. In the future, with only minor modifications, this tool could be integrated with the Introduction to Programming MOOC and used for giving automatic feedback to students about some aspects of their code, through the use of exercise-specific rules that check for predefined patterns.

As a part of the thesis we present eleven new rules. While these rules are still general-purpose (that is, they check for general good or bad programming practices), several of them are particularly suitable for first-year programming students, as they target mistakes that experienced programmers would never make.

B.2 The new rules

We are about to present the list of the new Eiffel Inspector rules. Notice how many of these are clearly oriented towards inexperienced programmers.

To name an example, rule CA030 checks for unnecessary unary sign operators. No experienced programmer would ever write '+3', if not in really exceptional cases. The only people who would reasonably write such an expression are first-year students, who might not be aware that the plus sign is redundant, or, in case they are, might erroneously think that this kind of redundancy is sometimes appreciated in programming. In fact, redundancy is not always a bad thing, however redundant unary operator are not considered a good practice.

Another example are naming conventions, which is something that first-year students have a strong tendency to overlook. Four of the new rules concern naming conventions.

For each of the new rules, at least one Eweasel test (see appendix C) has been created.

The list of rules follows here. For every rule, the name, severity, score and the official description (i.e. the text used in EiffelStudio) is reported, followed by a short explanation and an example.

B.2.1 CA030: Unnecessary sign operator

- **Name:** Unnecessary sign operator
- **Severity:** suggestion
- **Scope:** instruction
- **Description:** All unary operators for numbers are unnecessary, except for a single minus sign. They should be removed or the instruction should be checked for errors.

This rule checks for unnecessary unary sign operators, suggesting the removal of the redundant ones. The rule only applies to the sign of constants and will also work in the case of single constant values wrapped in one or more layers of parentheses. It does not apply to all types of expressions because Eiffel supports redefining operators. If the plus and minus operators are redefined, the assumptions we make for determining that they are unnecessary may no longer hold.

All the following lines trigger a violation of this rule:

Listing B.1: *Eiffel*: Sample violations of rule CA030

```
  -- Too many signs
l_int := +-2
l_real := -+4.1 -- Double violation by design
l_real := - -4.2
```

```
l_int := +(4)
l_int := -(-7)
l_real := -(-7.0)
l_int := -(0)

  -- Unnecessary single signs
l_int := +3
l_int := -0
l_real := +3.7
l_real := +0.0
```

The following lines will *not* trigger any violation:

Listing B.2: *Eiffel*: Example of code not violating rule CA030

```
l_int := +(-(-2 - 1)) -- Complex expression in parentheses
l_int := +l_int -- This rule only applies to literals
l_real := -0.0 -- -0.0 is not the same number as +0.0.
```

B.2.2 CA051: Empty and uncommented routine

- **Name:** Empty and uncommented routine
- **Severity:** warning
- **Scope:** feature
- **Description:** A routine which does not contain any instructions and has no comment too indicates that the implementation might be missing.

As a good practice, all features should be commented in Eiffel, and this is checked by the pre-existing CA036 rule. However, if the implementation of a routine is left empty for whatever reason, it becomes even more important to make sure that the feature is commented, explaining why is it fine that the implementation is blank.

this rule checks that all empty routines have a feature comment. If the comment is missing, the body of the routine is scanned, searching for a comment. If the routine body also contains no comments, a violation is triggered. The violation is a warning, thus has a higher severity than violations of rule CA036.

Sample violation:

Listing B.3: *Eiffel*: Sample violations of rule CA051

```
  -- This comment won't save us.
  -- There is no feature comment and no comment
  -- in the routine body.
no_op
  do
  end
```

The following routines will *not* trigger any violation:

Listing B.4: *Eiffel*: Example of code not violating rule CA051

```
no_op
    -- Do nothing.
  do
  end

reset
  do
    -- This object is stateless, no need to reset anything.
    -- Even if this feature has no proper feature comment,
    -- these comments within its body will save us from a
        violation.
  end
```

B.2.3 CA059: Empty 'rescue' clause

- **Name:** Empty 'rescue' clause
- **Severity:** warning
- **Scope:** feature
- **Description:** An empty rescue clause should be avoided and leads to undesirable program behaviour.

An empty 'rescue' clause in an Eiffel feature is useless, as it will not help handle exceptions in any way. It might however convey the wrong idea that that routine is able to handle exceptions, especially as EiffelStudio displays a special icon in the call stack next to routines having a rescue clause. For this reason, empty 'rescue' clauses should be always avoided.

Sample violation:

Listing B.5: *Eiffel*: Sample violation of rule CA059

```
do_something
    -- This feature should trigger a violation.
  do
    io.put_string ("Foo")
  rescue
  end
```

B.2.4 CA060: Inspect instruction has no 'when' branch

- **Name:** Inspect instruction has no 'when' branch
- **Severity:** warning
- **Scope:** instruction

- **Description:** 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.

There is never any point in an 'inspect' instruction having no 'when' branches. In such cases, if the 'inspect' instruction has an 'else' branch, then the instructions in it will always be executed, and if it has no 'else' branch an exception will always be thrown. In both cases, this should be avoided.

The violation message will be different depending on the presence or absence of the 'else' branch.

Sample violations:

Listing B.6: *Eiffel*: Sample violations of rule CA060

```
inspect l_foo
end

inspect l_foo
  else
    io.put_integer (0)
end
```

B.2.5 CA063: Class naming convention violated

- **Name:** Class naming convention violated
- **Severity:** warning
- **Scope:** class
- **Description:** Upholding naming conventions is one of the elements of a consistent coding style and enhances readability.

this rule checks that the Eiffel naming conventions for class names are respected. In particular, it checks that class identifiers are written in upper case and do not contain multiple consecutive underscore characters or trailing underscores.

All the following class identifiers violate this rule:

Listing B.7: *Eiffel*: Sample violations of rule CA063

```
non_uppercase
Not_ENTIRELY_UPPERCASE
DOUBLE__UNDERSCORE
TRAILING_UNDERSCORE_
```

B.2.6 CA064: Feature naming convention violated

- **Name:** Feature naming convention violated
- **Severity:** warning
- **Scope:** feature
- **Description:** Upholding naming conventions is one of the elements of a consistent coding style and enhances readability.

this rule checks that the Eiffel naming conventions for features are respected. In particular, it checks that feature identifiers are written in lower case and do not contain multiple consecutive underscore characters or trailing underscores.

All the following feature identifiers violate this rule:

Listing B.8: *Eiffel*: Sample violations of rule CA064

```
NON_LOWERCASE
Not_entirely_lowercase
double__underscore
trailing_underscore_
```

B.2.7 CA065: Local variable naming convention violated

- **Name:** Local variable naming convention violated
- **Severity:** warning
- **Scope:** feature
- **Description:** Upholding naming conventions is one of the elements of a consistent coding style and enhances readability.

this rule checks that the Eiffel naming conventions for locals are respected. In particular, it checks that local variable identifiers are written in lower case and do not contain multiple consecutive underscore characters or trailing underscores.

This rule can also check that local variable names start with the standard 'l_' prefix, as is common practice in Eiffel. This check is optional and can be enabled or disabled in the rule preferences. Even when the check is enabled, commonly used single-letter names ('i', 'j', 'k', 'n') are allowed.

All the following local variable identifiers violate this rule:

Listing B.9: *Eiffel*: Sample violations of rule CA065

```
l_bad__1: STRING
l_baD_2: STRING
l_baD_3: STRING
bad_4: STRING -- Acceptable if 'l_' prefix is not enforced
```

B.2.8 CA066: Argument naming convention violated

- **Name:** Argument naming convention violated
- **Severity:** warning
- **Scope:** feature
- **Description:** Upholding naming conventions is one of the elements of a consistent coding style and enhances readability.

this rule checks that the Eiffel naming conventions for arguments are respected. In particular, it checks that argument identifiers are written in lower case and do not contain multiple consecutive underscore characters or trailing underscores.

This rule can also check that argument names start with the standard 'a_' prefix, as is common practice in Eiffel. This check is optional and can be enabled or disabled in the rule preferences. Even when the check is enabled, commonly used single-letter names ('i', 'j', 'k', 'n') are allowed.

All the following argument identifiers violate this rule:

Listing B.10: *Eiffel*: Sample violations of rule CA066

```
feature_with_many_arguments (
    a_bad__1: STRING;
    a_baD_2: STRING;
    a_baD_3: STRING;
    bad_4: STRING; -- Acceptable if 'a_' prefix is not enforced
  )
```

B.2.9 CA079: Unneeded accessor function

- **Name:** Unneeded accessor function
- **Severity:** suggestion
- **Scope:** class
- **Description:** 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.

A violation of this rule is triggered by a function the body of which consists of a single assignment of the value of an attribute to 'Result'.

Unlike other programming languages, in Eiffel all attributes are not writable from outside the class. For this reason, in these situations it is generally better to expose the attribute directly and avoid a redundant getter function. The possibility that in future development the getter function is extended and needs

to implement some logic is still generally not a good justification, as in Eiffel converting an attribute to a function requires no changes to clients.

This mistake is typical of programmers landing to Eiffel from other programming languages, and in this case the attribute will most of the times be secret (not exported to anyone). However, this rule does not take the visibility of the attribute into account.

The following listing shows example of functions violating and not violating this rule.

Listing B.11: *Eiffel*: Sample violations of rule CA079

```
class MY_CLASS

feature -- Public

  get_secret_attribute_1: INTEGER
      -- Violation! This is a getter function.
    do
      Result := secret_attribute
    end

  get_secret_attribute_2: INTEGER
      -- No violation. We are performing some additional
          computation.
    do
      Result := secret_attribute + 1
    end

  get_secret_attribute_3: INTEGER
      -- No violation, same as above.
    do
      Result := secret_attribute
      io.put_string ("Foo")
    end

  get_secret_attribute_4: INTEGER
      -- No violation. Even if this is clearly equivalent
      -- to the first function, it does not match the simple
      -- getter pattern. There is clearly some problem here,
      -- but it's not up to us to guess what the programmer
      -- was trying to do.
    do
      Result := secret_attribute
      Result := secret_attribute
    end

feature {NONE} -- Secret area!

  secret_attribute: INTEGER

end
```

B.2.10 CA088: Mergeable feature clauses

- **Name:** Mergeable feature clauses
- **Severity:** hint
- **Scope:** class
- **Description:** Feature clauses with the same export status and comment could be possibly merged into one, or their comments could be made more specific.

This rule looks for 'feature' clauses having the same comment and the same export status within a single class, and raises a violation if any are found. There are some cases where this kind of duplication is intentional. For this reason, this rule only has the severity of 'hint'.

The rule is insensitive to case and whitespace in the comment text and the ordering of classes in the export list.

The following listing shows examples of mergeable feature clauses.

Listing B.12: *Eiffel*: Sample violations of rule CA088. Comments explain what feature clauses are considered duplicates and what not.

```
-- #1:
feature -- Public

  feature_1: INTEGER

  feature_2: INTEGER

-- #2:
feature --          public
  -- Duplicate of #1

  feature_3: INTEGER

-- #3:
feature --       PuBlIc
  -- Duplicate of #1

-- #4:
feature -- Different section

-- #5:
feature {ARGUMENTS, TEST} -- Restricted

  feature_4: INTEGER

-- #6:
feature {TEST, ARGUMENTS} --     restricted
  -- Duplicate of #5

-- #7:
feature {TEST, ARGUMENTS} -- Different section

-- #8:
feature {  TEST,  ARGUMENTS } --Restricted
```

```
-- Duplicate of #5

feature_5: INTEGER
```

B.2.11 CA089: Explicit redundant inheritance

- **Name:** Explicit redundant inheritance
- **Severity:** suggestion
- **Scope:** class
- **Description:** Explicitly duplicated inheritance links are redundant if there is no renaming, redefining or change of export status. One should be removed.

This rule raises a violation if a class explicitly declares the same parent twice in the 'inherit' clause without any renaming, redefining or change of export status, which is redundant. In doing this, no distinction is made between conforming and non-conforming inheritance.

The following listing shows an example of violation of this rule:

Listing B.13: *Eiffel*: Sample violations of rule CA089.

```
class
  MY_CLASS

inherit
  PARENT_CLASS
  OTHER_PARENT_CLASS
  PARENT_CLASS

end
```

Appendix C

Eiffel inspector: support for testability of rules

Besides the implementation of the new code analysis rules, a contribution of this thesis is the extension of the Eweasel unit testing tool so that it supports unit testing of code analysis rules. This also required some changes in Eiffel Inspector itself. In this appendix, we report on the changes on both tools.

C.1 Eweasel

Eweasel [10] is a unit testing tool for the Eiffel compiler, used at Eiffel Software to make sure that changes and new implementations in the Eiffel compiler do not introduce regressions or break existing code.

Eweasel can run tests in batch. Tests are organized in different categories, depending on the features of the compiler that they are supposed to test (for example, parsing, incremental compilation, etc.).

C.1.1 Tests

Every test is defined by a test control file. The control file contains a sequence of commands in a simple ad-hoc language.

Most tests consist of creating a test project in an output folder, attempting to compile it, possibily run it, and check that the output from the compiler and from the executable file matches the expected output. For doing this, some auxiliary files are also needed (typically the class files and the project file of the project to be compiled).

The following example, taken from a real test for the Eiffel compiler (syntax002) should make it immediately clear:

```
1 -- This is a test control file
2
```

```
3  test_name no-index-tag
4  test_description Index_clause without an Index_tag
5  copy_sub Ace $TEST Ace
6  copy_raw test.e $CLUSTER test.e
7  compile_melted
8  compile_result syntax_warning TEST 8
9  test_end
```

Commands on lines 3 and 4 specify the test name and description.

On line 5, command 'copy_sub' requests file 'Ace' (the Eiffel project file, which is a part of this sample test) to be copied to the output directory ($TEST). In doing this, the content of the Ace file is also modified, by substituting every reference to any environment variable within its body with the actual value of the variable when running Eweasel.

Command 'copy_raw', on line 6, copies the 'test.e' class file to the output directory ($CLUSTER is generally a subfolder of the output directory) without applying any substitution.

Command 'compile_melted' on line 7 launchs the compiler on the output project.

Command 'compile_result' checks that the actual compiler output corresponds to the expected output (which in this case is a syntax warning on line 8 of class 'TEST'). This command can be considered an assertion. If the compilation output corresponds to the expected output, the assertion holds and execution continues to the next line. If it doesn't, an error is generated and the execution of the test interrupted.

If no error is raised by the preceding commands, command 'test_end' is eventually reached and the test terminates successfully.

It is important to note that the Eiffel compiler is simply invoked from the command line. This means that the output of the compiler must be manually parsed by Eweasel before being compared to to the expected one.

C.2 Eiffel Inspector support for Eweasel

As the Eiffel Inspector tool, after being developed as a Master's thesis project [17], was merged into the EiffelStudio trunk, it became crucial to have a way for running unit tests and regression tests on code analysis rules.

A typical test for an Eiffel Inspector rule should generally consist of a class file including some code violating the rule and other code not violating it. The code should be analyzed disabling all rules but the one that should be tested, and it should be checked that the raised violations match the expectation. This is very similar to Eweasel tests in principle, therefore extending Eweasel for supporting this kind of tests was a natural choice.

The most notable additions to Eweasel that were necessary were the 'analyze_code' and 'analysis_result' command. In principle, these are analogous to

the 'compile_*' and 'compile_result' instructions, however they clearly require different arguments and perform a different parsing of the output.

An example is the most effective way to give an idea. The following listing shows a test control file for an Eiffel Inspector test:

```
1  -- This is a test control file
2
3  test_name ca-forced-rules-preferences
4  test_description Test forcing rules and preferences from the
       command line.
5  copy_sub Ace $TEST Ace
6  copy_raw test.e $CLUSTER test.e
7
8  -- Non-existent preference
9  analyze_code rule "CA043 (Depth threshold=3, Fake preference
       =9); CA080"
10 analyze_code_result violation "TEST CA043 CA080"
       preference_warning
```

Up to line 6, everything is substantially the same as in a standard test control file. On line 9 we find the first new command, 'analyze_code'. This command will first compile the target project and then, if compilation is successful, run the code analysis on it. Even though the names are slightly different, this commands supports all the command line switches implemented by Eiffel Inspector, which can be found in the Eiffel Inspector online documentation [7].

On line 10, command 'analyze_code_result' checks that the analysis result matches the expected results. In this example, the expected result consists of two violations, respectively of rules CA043 and CA080, both of them in class TEST. The lines where the violations are expected to occur can also specified (by writing 'CA080:23'), however this is optional. If no line number is specified for a certain expected violation, as in this example, Eweasel will 'accept' it regardless of the location where it appears.

The expected result can also include other errors reported by Eiffel Inspector, such a class or rule not being found. In our example, the 'preference_warning' keyword signifies that Eiffel Inspector is expected to throw a warning about a non-existing rule preference name being specified. Eweasel is now able to parse such warnings, allowing to test that Eiffel Inspector throws the correct warnings or errors when called with incorrect arguments.

C.3 Changes and improvements to Eiffel Inspector

The implementation of support for rule testability also required some changes to Eiffel Inspector itself, mostly related to the command line support. We made changes to the formatting and the information included in the output messages in order to achieve better readability and parsability.

The most notable change is probably the addition of the new '-caforcerules'

command line switch. This makes it possible to specify a list of rules to be used for the analysis directly from the command line. Before the implementation of this switch, the only way for selecting the rules to be used from the command line was using the '-caloadprefs' switch, which loads an XML preference file. This was however very inconvenient, as XML preference files are expected to explicitly list all the supported rules, so this approach required to maintain a large preference file for every single test.

The new '-caforcerules' switch supports specifying preferences directly from the command line for those rules supporting special preferences.

C.3.1 Eiffel Inspector as an automated feedback generator for programming exercises

It is worth noting that these changes, although not major, take Eiffel Inspector one step closer to being ready for being used for analyzing code submitted by students in programming exercises.

Eiffel Inspector already provides a flexible framework for rule and pattern checking in the code. Its main characteristic which doesn't fit well the purpose of student code assessment is that rules are assumed to be general, i.e., applicable to any program. On the other hand, automated code assessment often requires to check some rules that are specific for the exercise being checked (such as a maximum number of allowed locals, or forbidding the use of a particular instruction or class). However, the ability of invoking Eiffel Inspector conveniently selecting and parametrizing the rules to be enabled directly from the command line, makes it possible to create special code analysis rules which are normally kept disabled and are only enabled when processing the exercises which they apply to.

Further modifications to Inspector Eiffel have the potential of making it very easy to integrate into the Introduction to Programming MOOC without the need to write a new separate tool, which would eventually mostly overlap with Eiffel Inspector's functionality.

C.3.2 Additional contributions

While working on Eiffel Inspector, some other minor enhancements or bug fixes have been made. Additionally, Eiffel Inspector has been run on the source code of AutoTeach during its development, with the purpose of improving code quality. Besides highlighting code quality issues in AutoTeach, this process also brought to light some bugs or issues in some of the code analysis tools. We have fixed some of these straight away, for all the others we have created an Eweasel test highlighting the incorrect behavior, so that the Eiffel developers could address the issue.

Bibliography

[1] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. "Plaggie: GNU-licensed Source Code Plagiarism Detection Engine for Java Exercises". In: *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*. Baltic Sea '06. Uppsala, Sweden: ACM, 2006, pp. 141–142. DOI: 10.1145/1315803.1315831. URL: http://doi.acm.org/10.1145/1315803.1315831.

[2] Alex Aiken. *Moss: Measure Of Software Similarity*. 1994. URL: http://theory.stanford.edu/~aiken/moss/.

[3] Armando Fox. "From MOOCs to SPOCs". In: *Commun. ACM* 56.12 (Dec. 2013), pp. 38–40. ISSN: 0001-0782. DOI: 10.1145/2535918. URL: http://doi.acm.org/10.1145/2535918.

[4] J. B. Hext and J. W. Winings. "An Automatic Grading Scheme for Simple Programming Exercises". In: *Commun. ACM* 12.5 (May 1969), pp. 272–275. ISSN: 0001-0782. DOI: 10.1145/362946.362981. URL: http://doi.acm.org/10.1145/362946.362981.

[5] Petri Ihantola et al. "Review of Recent Systems for Automatic Assessment of Programming Assignments". In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling '10. Koli, Finland: ACM, 2010, pp. 86–93. ISBN: 978-1-4503-0520-4. DOI: 10.1145/1930464.1930480. URL: http://doi.acm.org/10.1145/1930464.1930480.

[6] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. "Automated Feedback Generation for Introductory Programming Assignments". In: *SIGPLAN Not.* 48.6 (June 2013), pp. 15–26. ISSN: 0362-1340. DOI: 10.1145/2499370.2462195. URL: http://doi.acm.org/10.1145/2499370.2462195.

[7] Eiffel Software. *Eiffel Inspector command line documentation*. URL: https://docs.eiffel.com/book/eiffelstudio/eiffel-inspector-running-analyzer#Command_Line (visited on 09/16/2014).

[8] Eiffel Software. *Eiffel language*. URL: http://www.eiffel.com.

[9] Eiffel Software. *EiffelStudio command line documentation*. URL: https://docs.eiffel.com/book/eiffelstudio/eiffelstudio-using-command-line-options (visited on 09/17/2014).

[10] Eiffel Software. *Eweasel*. URL: https://dev.eiffel.com/Eweasel (visited on 09/16/2014).

[11] Chair of Software Engineering. *EVE (Eiffel Verification Environment)*. URL: http://se.inf.ethz.ch/research/eve/.

[12] UHS. *Universal Hint System*. 1988. URL: http://www.uhs-hints.com/ (visited on 09/17/2014).

[13] Milena Vujošević-Janičić et al. "Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments". In: *Inf. Softw. Technol.* 55.6 (June 2013), pp. 1004–1016. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2012.12.005. URL: http://dx.doi.org/10.1016/j.infsof.2012.12.005.

[14] Wikipedia. *Garbage in, garbage out (GIGO principle)*. 2014. URL: http://en.wikipedia.org/wiki/http://en.wikipedia.org/wiki/Garbage_in,_garbage_out (visited on 09/17/2014).

[15] Wikipedia. *Kleene logic*. 2014. URL: http://en.wikipedia.org/wiki/Three-valued_logic#Kleene_logic (visited on 09/14/2014).

[16] Wikipedia. *Three-valued logic*. 2014. URL: http://en.wikipedia.org/wiki/Three-valued_logic (visited on 09/14/2014).

[17] Stefan Zurfluh. "Rule-Based Code Analysis". MA thesis. ETH Zurich, 2014.