21

Pattern Wizard

The main goal of this thesis was to provide the reusable Eiffel components corresponding to the design patterns of [Gamma 1995] found componentizable — given the Eiffel language's facilities and advanced mechanisms such as Design by Contract™, genericity, multiple inheritance, and agents.

This examination of the patterns listed in *Design Patterns* revealed that some patterns can be transformed more or less easily into reusable libraries whereas other patterns resist any componentization attempt. The latter require content-dependent information, which can only be given by the programmer. Even though it was not possible to provide a reusable component in such cases, I still wanted to help developers as much as possible and built a tool that would take care of the repetitive tasks automatically. Hence the development of the Pattern Wizard.

See "Definition: Componentization", page 26.

This chapter gives a tutorial about how to use the tool and take advantage of it. Then, it describes the design and implementation of the wizard, and discusses its limitations. Finally, it presents some related work.

21.1 WHY AN AUTOMATIC CODE GENERATION TOOL?

A design pattern is a solution to a particular design problem but it is not code itself. Programmers must implement it anew whenever they want to apply the pattern. Componentization provides a solution to this problem but unfortunately not all design patterns are componentizable. Thus, programmers still need to implement the code for some patterns. This is the point where an automatic code generation tool comes into play. Some developers, in particular newcomers, may have difficulties to implement a design pattern from just a book description, even if there are some code samples. Others simply may find it tedious to implement the patterns because it is repetitive: it is always the same kind of code to write afresh for each new development. Hence the interest of the Pattern Wizard.

The Pattern Wizard may also be interesting for the componentizable patterns for at least two reasons:

- The pattern is not fully componentizable and the componentized version cannot handle the given situation.
- The reusable component is applicable but not desirable because of performance reasons for example (e.g. in embedded systems).

  Section 9.3 showed that using the Visitor Library on the Gobo Eiffel Lint tool results in a performance overhead (less than twice as slow) compared to a traditional implementation of the *Visitor* pattern. Therefore it may be impossible to use the Visitor Library in some application domains that require

See "Gobo Eiffel Lint with the Visitor Library", page 138.

topmost performance. Thus it would be interesting to extend the Pattern Wizard to support the *Visitor* pattern to have better code performance when it is needed.

The next section gives a tutorial of the Pattern Wizard that already supports all non-componentizable patterns for which it is possible to generate skeleton classes. The next implementation step will be to extend the wizard to support componentizable design patterns (and possibly other target programming languages).

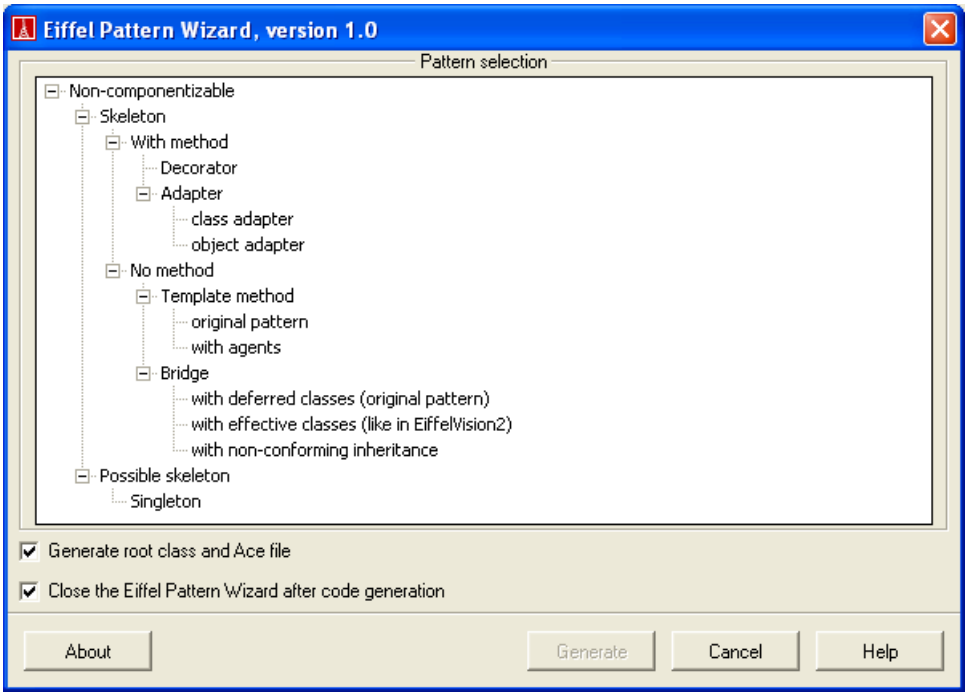
See "Design pattern componentizability classification (filled)", page 90.

21.2 TUTORIAL

Before moving to the design and implementation of the Pattern Wizard, it is interesting to have a look at the actual product. This section explains how to use the wizard to generate code for the *Decorator* pattern; then, it shows briefly the graphical interfaces for the other supported patterns.

Example of the Decorator pattern

When launching the Pattern Wizard, the first window that shows up is the following:



Initial window of the Pattern Wizard

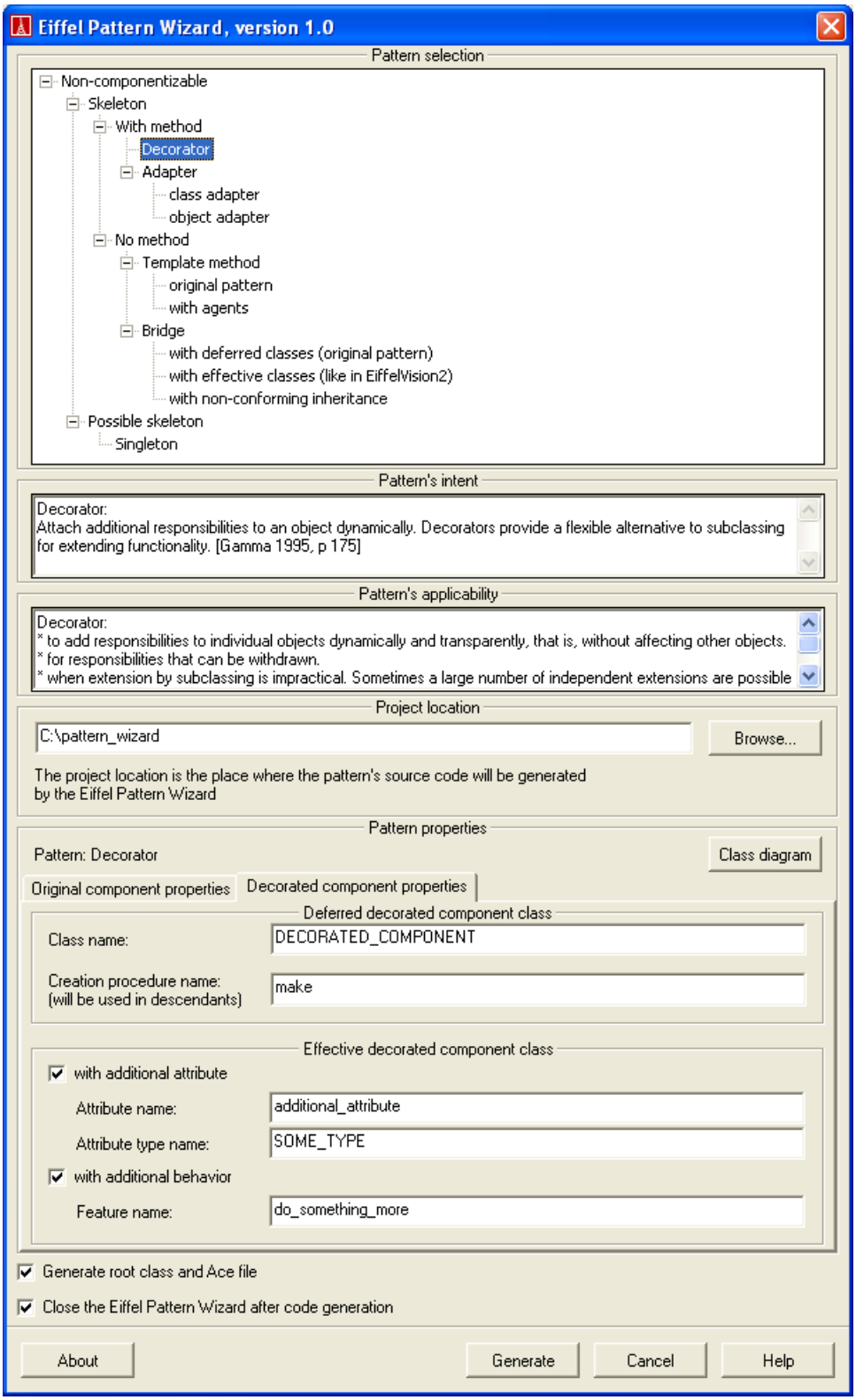
The tree view enables you to select the pattern you want to generate code for. This tree view recalls the pattern componentizability classification described in chapter 6. Not all items are selectable; for example, clicking on "Possible skeleton classes" will have no effect. You need to click on actual pattern names like "Singleton", "Decorator", and so on, namely on the end tree items, not on tree nodes. Selecting a pattern name will make the bottom part of the window to change and show pattern-specific information. The "Generate" button will also be enabled.

The toggle buttons at the bottom enables you to say whether you want the wizard to generate a whole Eiffel project, meaning the pattern classes plus a root class and an Ace file. You can also decide to close the wizard after code generation if you need to generate code for only one pattern.

At any time, you can consult the online help by clicking the "Help" button on the bottom right-hand side of the window. It will open a PDF file that recalls the information contained in this chapter.

The "About" button gives access to some general information about the Pattern Wizard (product version, contact information, etc.).

Let's suppose we select the pattern *Decorator*. The initial window will be extended to display information and properties that are specific to the *Decorator* pattern. The corresponding window appears below:

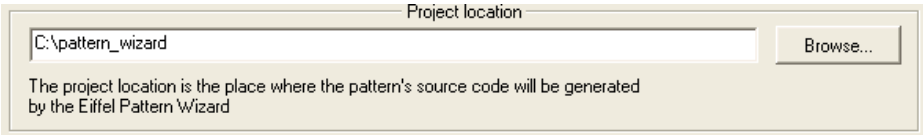


Pattern Wizard window once the Decorator pattern has been selected

- The first two extra boxes display the pattern's intent and applicability. This information is taken from the *Decorator* chapter of *Design Patterns*. It is pure information whose goal is to help the user know whether this pattern is of interest to his problem. It does not intervene in the code generation process.

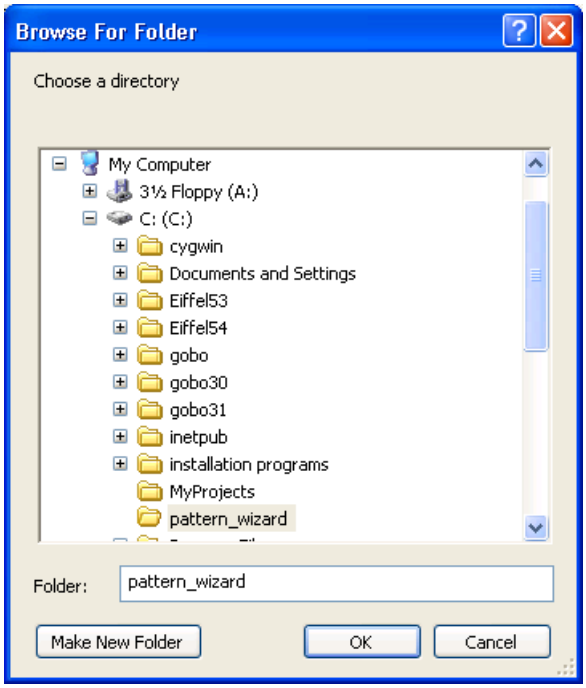
[Gamma 1995], *p 175-184.*

- The subsequent box enables you to select the project directory, namely the folder where the code will be generated.



Selection of the project directory

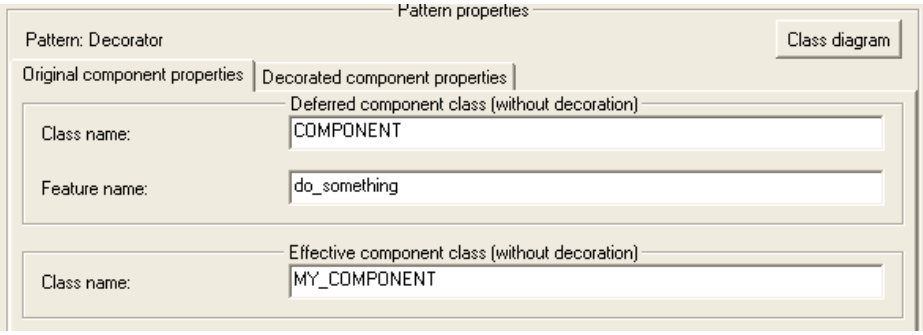
You can either write the directory path in the text field on the left or select a directory by clicking the "Browse..." button. It will open a modal dialog:



Dialog to select a project directory folder

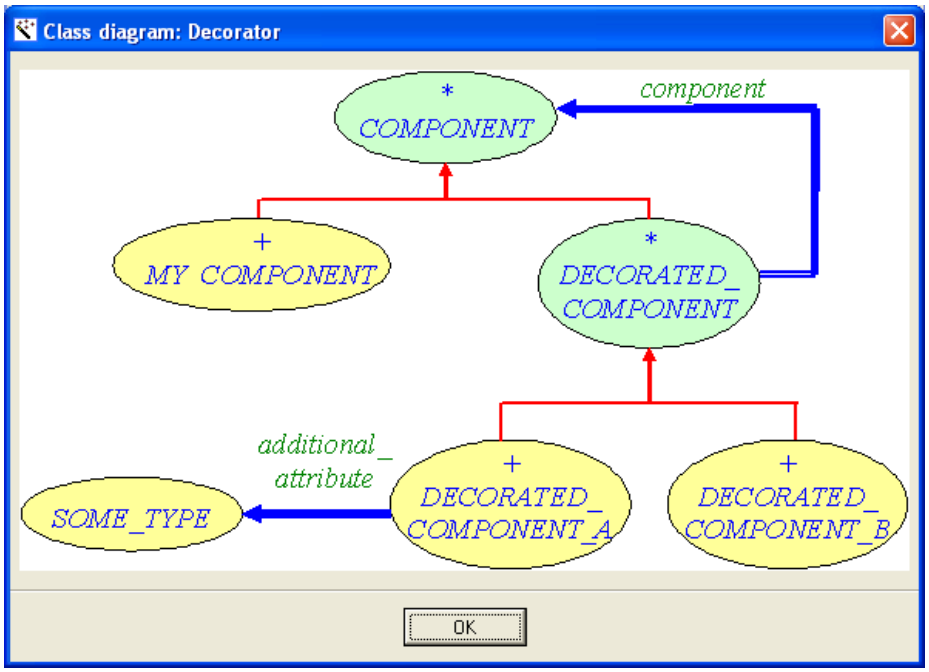
It shows the file hierarchy on your computer and enables you to select either an existing directory or create a new one at the place you want. By default, the Pattern Wizard creates a folder "pattern_wizard" under your C drive and use it as project directory. You can choose to use this default directory; in that case, just leave the "Project location" box unchanged.

- The next box corresponds to the pattern-specific properties you can select; they are the parameters you can set for the code generation.



Frame to select the Decorator properties (first tab: original component properties)

To make your job easier, the Pattern Wizard gives you the possibility to have a look at the class diagram of a typical application using the chosen pattern, here the *Decorator*. Simply click the "Class diagram" button on the top right.

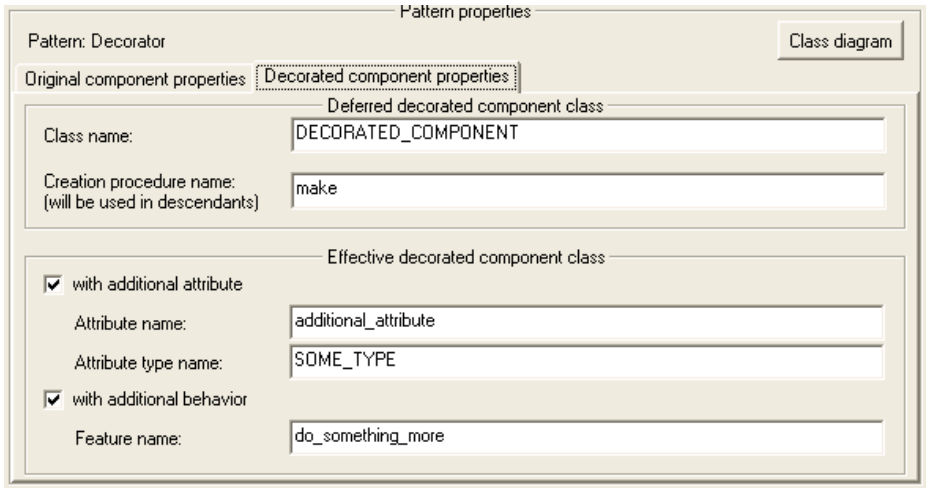


Class diagram of a typical application using the Decorator pattern

You can see that there are two class hierarchies: one for the component classes and a second one for the decorated component classes. They are represented by two tabs in the "Pattern properties" frame.

The first tab concerns the component classes: the deferred class and the effective descendant class (*COMPONENT* respectively *MY_COMPONENT* in the example class diagram). You can also specify the name of the feature that will appear in the parent class. Again, you can choose to rely on the defaults, in which case you don't need to change anything.

Let's have a look at the second tab now, which concerns the decorated classes:



Frame to select the Decorator properties (second tab: decorated component properties)

First, you can choose the name and creation procedure name of the parent class of the decorated components, called *DECORATED_COMPONENT* in the previous class diagram.

Then, you can choose what kind of effective decorated components you want, either with an additional attribute (of which you can choose the name and type) or with an additional procedure (of which you can choose the name) or both. Simply select the toggle buttons "with additional attribute" and "with additional behavior" accordingly. By default, both check boxes are selected. If you unselect one of them, the relative text fields and labels will be disabled.

If you choose to have a decorated component with an additional attribute, the Pattern Wizard will generate a new class corresponding to the attribute's type no matter whether it corresponds to an existing class or not. Therefore it may be that the generated code does not compile (because of this extra class). You will need to adapt the generated Ace file to use your existing class and not the generated one.

Once you have chosen the pattern properties (you can also leave them unchanged and rely on the default values), you can click the "Generate" button at the bottom of the window, which will launch the code generation.

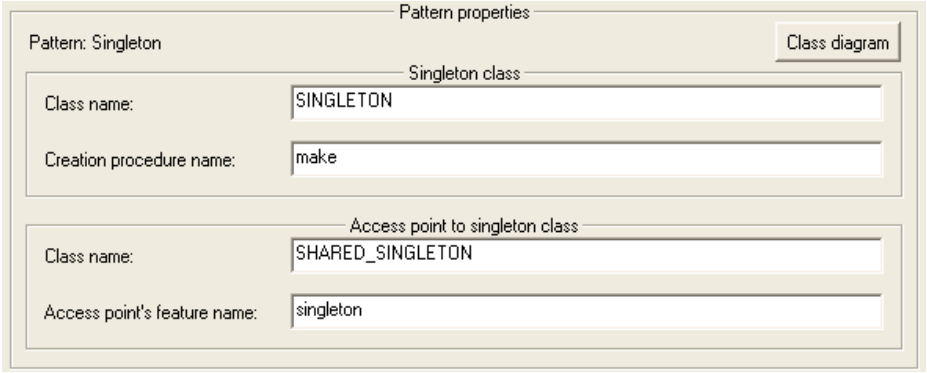
If you asked the wizard to close after code generation, clicking "Generate" will also close the wizard's window unless a problem occurs during the code generation (because of invalid inputs). If you didn't check the box "Close the pattern after code generation", the window will not be closed; the wizard will display a message saying that the code generation was successful:

Message after a successful code generation

Other supported patterns

The Pattern Wizard supports four other patterns (and variants): the *Singleton*, *Adapter*, *Template method*, and *Bridge* design patterns. This tutorial does not explain in detail how to use the wizard for each pattern because the approach resembles very much what we just did for the *Decorator* pattern. It just shows the "Pattern properties" frame for each pattern and explains the particularities, if any.

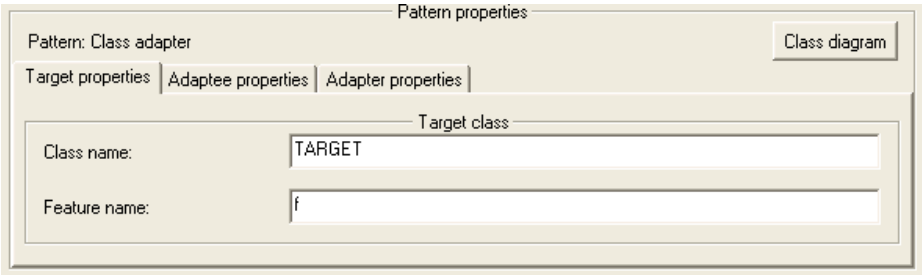
- *Singleton*:



"Pattern properties" frame for the Singleton pattern

You can select the name of the *Singleton* class and the name of its point of access. You can also choose the creation procedure of the *Singleton* class and the name of the query that will return the *Singleton* instance in the access point class. Please refer to chapter 18 for more information about the *Singleton* pattern.

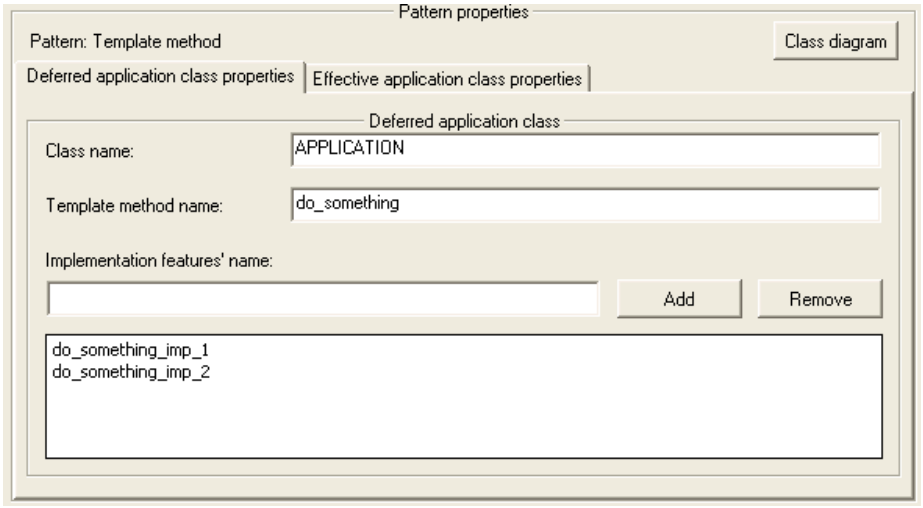
- *Adapter*:



"Pattern properties" frame for the Adapter pattern

You can choose: the name of the target class (the one used by clients) and the name of the feature it exposes; the name of the adaptee class and the name of the feature it declares (the one we want to use in the implementation of the adapter feature); the name of the adapter class (that reconciles the interfaces of the target and adaptee classes). The wizard supports both class and object versions of the *Adapter*. Please refer to section 16.2 for more information.

- *Template method*:

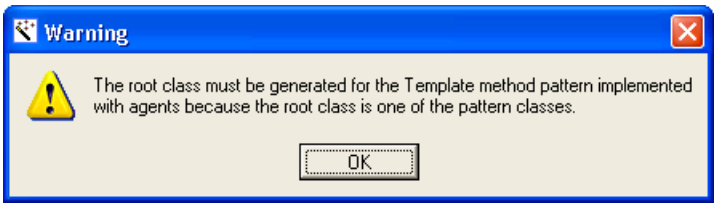


"Pattern properties" frame for the Template method pattern

A "Template method" is basically a feature whose implementation is defined in terms of other features (the implementation features), which are deferred and effected in descendant classes. The wizard's graphical interface for the *Template method* pattern enables you to choose the different class and feature names.

The Pattern Wizard supports two variants of this pattern: the original pattern version, which I just described, and a version using agents. Both variants are described in section 17.1 of this thesis.

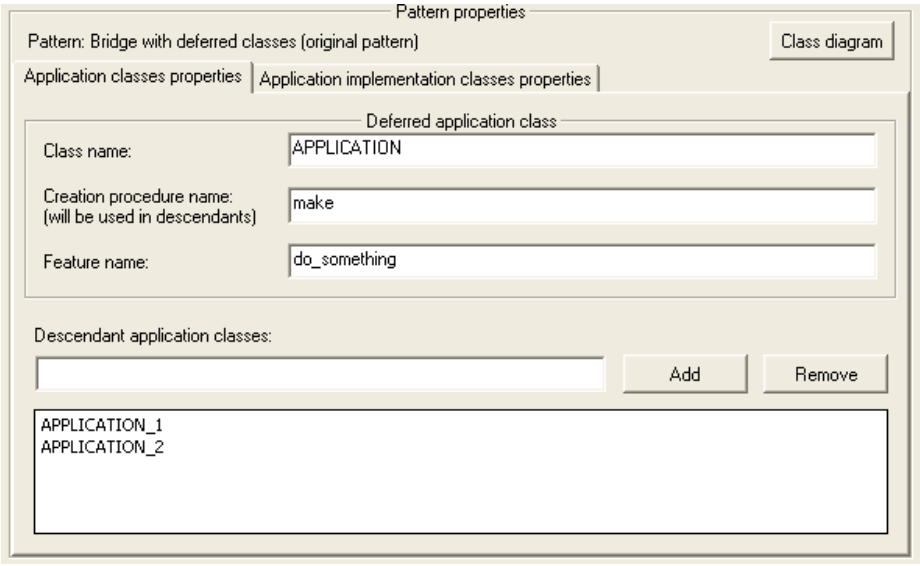
A particularity of the version implemented with agents is that the root class is one of the pattern classes. Therefore it is compulsory to select the option "Generate root class and Ace file" at the bottom of the pattern's window. If you don't select it and click the "Generate" button, you will get a warning message:



Warning message for the Template method with agents pattern

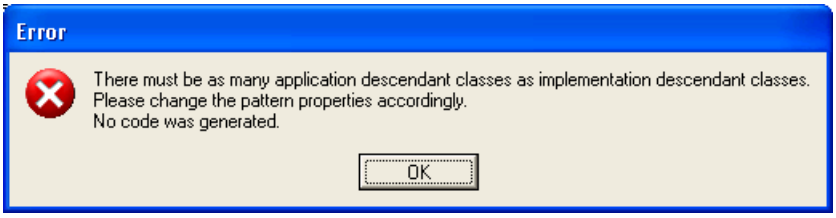
and the wizard will automatically select the option for you and generate the correct code.

- *Bridge*:



"Pattern properties" frame for the Bridge pattern

The *Bridge* pattern relies on two parallel hierarchies: the application classes and the implementation classes. The Pattern Wizard enables you to select the name of all involved classes and features. For example, you can choose the name of the application class's descendants and of the implementation class's descendants. One constraint is that you must have as many descendants of the application class as descendants of the implementation class. If it is not the case and you click the "Generate button", you will get an error message:



Error message in case of invalid input for the Bridge pattern

No code will be generated.

The Pattern Wizard supports three variants of the *Bridge* pattern: the original pattern, a version using effective classes only, and a third variant using non-conforming inheritance. All three variants are described in section 17.2.

21.3 DESIGN AND IMPLEMENTATION

The Pattern Wizard automatically generates Eiffel classes — and possibly a project root class and an Ace file — that programmers will have to fill in to build their systems. The code generation relies on template files with placeholders that the wizard fills in with the pattern properties entered by the user. Let's have a closer look at the design and implementation of the Pattern Wizard.

The notions of root class and Ace file are described in appendix A.

Objectives

The Pattern Wizard targets the non-componentizable patterns of categories 2.1 and 2.2 of the componentizability classification appearing in section 6.3 for which it is possible to generate skeleton classes (i.e. Eiffel classes that compile and capture the entire pattern structure but miss implementation that developers will have to provide). The idea is both to simplify the job of programmers by preparing the code and to ensure the design pattern gets implemented correctly. Five design patterns belong to the categories 2.1 and 2.2, some of them having several variants:

See "Design pattern componentizability classification (filled)", page 90.

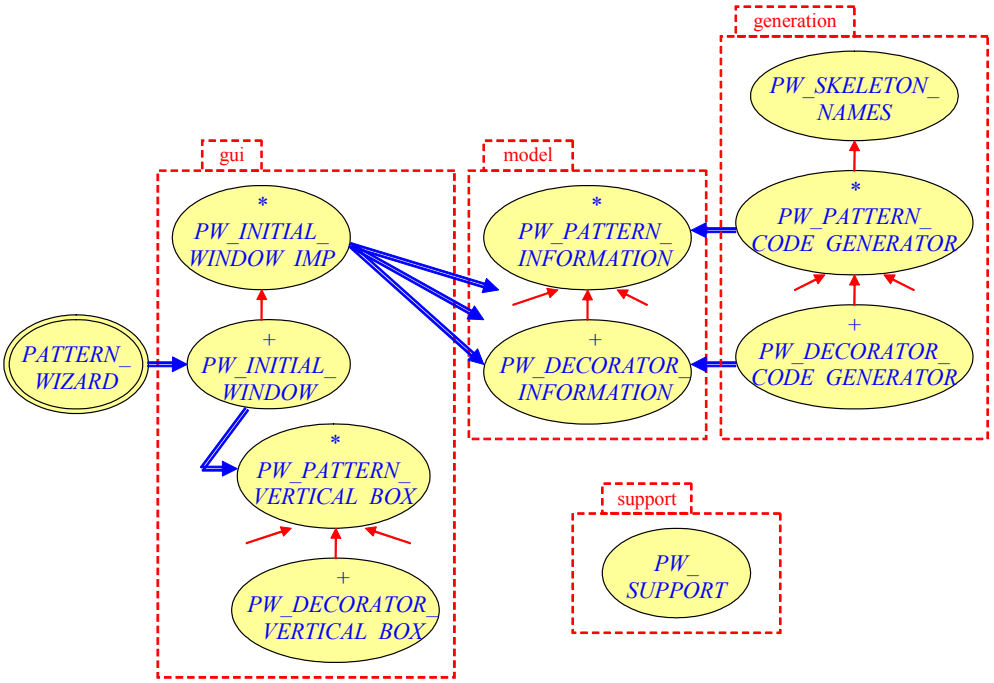
- *Adapter* (*Class adapter* and *Object adapter*)
- *Decorator*
- *Template Method* (original pattern and variant implementation using agents)
- *Bridge* (original pattern using deferred classes, variant using effective classes only, and an implementation using non-conforming inheritance)
- *Singleton*

The Pattern Wizard has been carefully designed to:

- Separate the underlying model (pattern information, code generation) and the GUI parts: the corresponding classes appear in different clusters (see Overall architecture).
- Enforce reusability: motifs appearing several times in the variant windows of the Pattern Wizard have been captured into reusable components to avoid code repetition.
- Ensure extensibility: the Pattern Wizard can easily be extended to support other design patterns. (I will explain more about that after presenting the application's architecture.)

Overall architecture

The following class diagram shows the overall architecture of the Pattern Wizard. For simplicity, it does not show all the classes. For example, it only shows the classes (GUI, model, and code generation) corresponding to the *Decorator* pattern; you have to imagine the counterparts for the other supported patterns.



Simplified class diagram of the Pattern Wizard application

The Pattern Wizard classes are grouped into four main clusters:

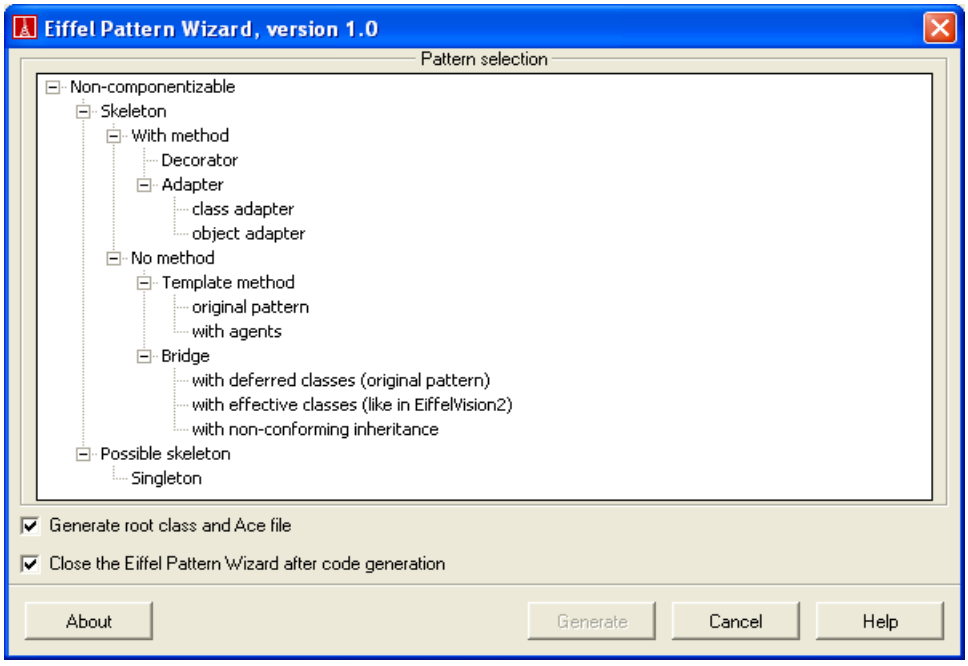
- "gui": This cluster contains all GUI-related classes. It has a subcluster "components" for all reusable GUI components mentioned before (frames, horizontal and vertical boxes, etc.). The classes that do not belong to the subcluster "components" correspond to pattern-specific GUI components and windows of the Pattern Wizard.
- "model": This cluster includes the class *PW_PATTERN_INFORMATION* and its descendants, which contain the information needed to generate code for each pattern.

- "generation": This cluster contains the class *PW_PATTERN_CODE_GENERATOR* and its descendants, which take care of the actual code generation based on the *PW_PATTERN_INFORMATION* classes and the placeholder names defined in the class *PW_SKELETON_NAMES*.
- "support": This cluster contains the helper class *PW_SUPPORT* that contains useful features like *pattern_delivery_directory*, *directory_exists*, and *file_exists*.

Graphical User Interface

The class *PW_INITIAL_WINDOW* corresponds to the first window that appears when launching the *PATTERN_WIZARD* application (reproduced below).

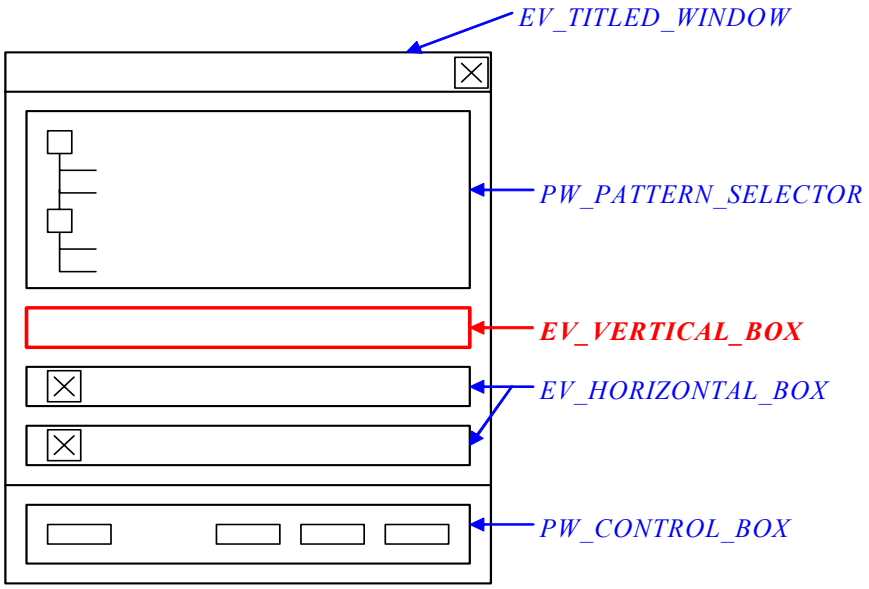
This window already appeared on page 324.



Initial window of the Pattern Wizard

It consists of a tree view of the supported patterns plus a few controls. When the user selects a pattern in the tree view, the bottom part of the window changes and shows pattern-specific information and properties the user has to enter (unless he wants to rely on the default values).

Here is the widget layout of this initial window of the Pattern Wizard that permits such dynamic transformation:



Widget layout of the Pattern Wizard's initial window

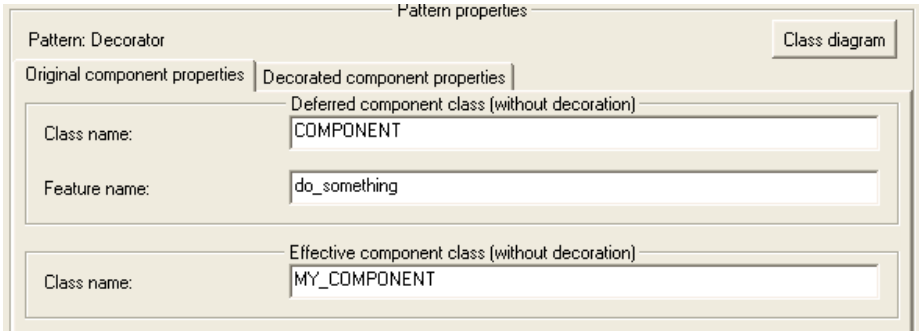
Each tree item is associated with an action *select_pattern*, which creates an instance of a pattern-specific descendant of *PW_PATTERN_VERTICAL_BOX* and extend the vertical box in red in the above figure with it.

The *PW_PATTERN_VERTICAL_BOX* displays some information — the patterns' intent and applicability — directed at the user to help him know whether the selected design pattern is useful for problem. Besides, it shows the pattern properties that can be changed before the code generation; for example, the name of classes and features of those classes.

Model

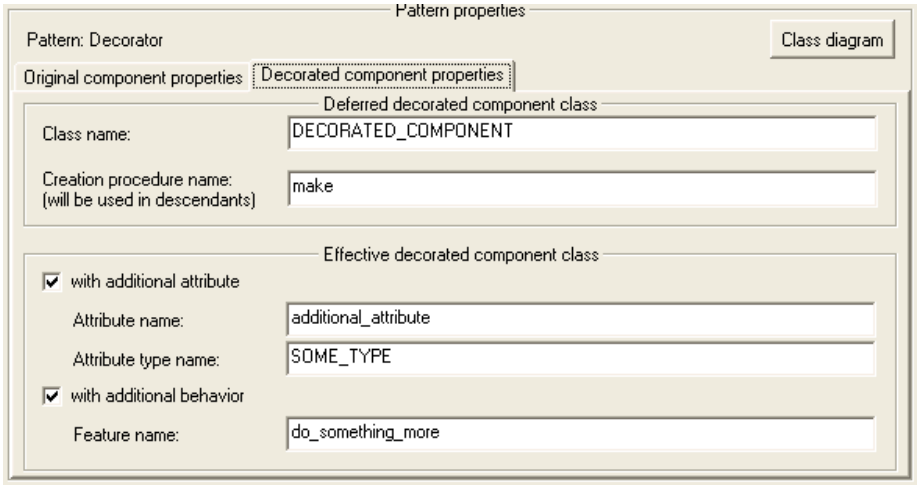
The cluster "model" is composed of the class *PW_PATTERN_INFORMATION* and its descendants. They contain the information the user can enter in the different text fields and other controls of the Pattern Wizard's GUI, which will be used by the *PW_PATTERN_CODE_GENERATOR*.

Let's take the example of the *Decorator* pattern. The pattern properties frame looks like this:



Frame to select the Decorator properties (first tab: original component properties)

The second tab with the properties of the decorated component appears next:



Frame to select the Decorator properties (second tab: decorated component properties)

The model was designed as a "repository" of information given by the user via the wizard's GUI. There is a direct mapping between the two. For example, the field "Class name" of the "Original component properties" tab is represented by an attribute *component_class_name* in the class *PW_DECORATOR_INFORMATION*; the field "Creation procedure name" in the "Decorated component properties" tab is modeled by an attribute *decorated_component_creation_procedure_name*.

Each attribute has a corresponding setter procedure to make it possible for the GUI classes to construct the *PW_PATTERN_INFORMATION* from the information entered by the user. This is done in the class *PW_INITIAL_WINDOW_IMP*.

The function *decorator_info* is sketched below:

```
deferred class
        PW_INITIAL_WINDOW_IMP
...
feature {NONE} -- Implementation (Pattern information)

        decorator_info: PW_DECORATOR_INFORMATION is
                        -- Selected information about the chosen pattern
                require
                        decorator_pattern_vbox_not_void: decorator_pattern_vbox /= Void
                local
                        frame: PW_DECORATOR_PROPERTY_SELECTOR
                do
                        create Result
                        frame := decorator_pattern_vbox.pattern_properties_frame
                        Result.set_component_class_name (frame.component_class_name)
                        Result.set_feature_name (...)
                        Result.set_effective_component_class_name (...)
                        Result.set_decorated_component_class_name (...)
                        Result.set_decorated_component_creation_procedure_name (...)
                        if frame.is_component_with_additional_attribute_generation then
                                Result.set_component_with_
                                                additional_attribute_generation (True)
                                Result.set_additional_attribute_name (...)
                                Result.set_additional_attribute_type_name (...)
                        end
                        if frame.is_component_with_additional_behavior_generation then
                                Result.set_component_with_
                                                additional_behavior_generation (True)
                                Result.set_additional_feature_name (...)
                        end
                ensure
                        decorator_info_not_void: Result /= Void
                end
...
end
```

Construction of a PW_DECORATOR_INFORMATION

The class *PW_PATTERN_INFORMATION* also exposes a query *is_complete*, which permits to know whether all information has been filled by the user; *is_complete* must be true before any code generation.

Generation

The cluster “generation” contains the class *PW_PATTERN_CODE_GENERATOR* and its descendants (one descendant per pattern). Here is the interface of the class *PW_PATTERN_CODE_GENERATOR*:

```
deferred class interface
        PW_PATTERN_CODE_GENERATOR

feature -- Access

        pattern_info: PW_PATTERN_INFORMATION
                        -- Pattern information needed for the code generation
                        -- (name of classes, name of features, etc.)

        project_directory: STRING
                        -- Path of the project directory (where the code will be generated)
```

Interface of the class PW_PATTERN_CODE_GENERATOR

```
feature -- Status report

        root_class_and_ace_file_generation: BOOLEAN
                -- Should a root class and an Ace file be generated?

feature -- Element change

        set_pattern_info (a_pattern_info: like pattern_info)
                -- Set pattern_info to a_pattern_info.
            require
                a_pattern_info_not_void: a_pattern_info /= Void
            ensure
                pattern_info_set: pattern_info = a_pattern_info

        set_project_directory (a_project_directory: like project_directory)
                -- Set project_directory to a_project_directory.
                -- Add '\' at the end if none.
            require
                a_project_directory_not_void: a_project_directory /= Void
                a_project_directory_not_empty: not a_project_directory.is_empty
                directory_exists: directory_exists (a_project_directory)
            ensure
                project_directory_set: project_directory /= Void and then
                                       not project_directory.is_empty

        set_root_class_and_ace_file_generation (
            a_value: like root_class_and_ace_file_generation)
                -- Set root_class_and_ace_file_generation to a_value.
            ensure
                root_class_and_ace_file_generation_set:
                    root_class_and_ace_file_generation = a_value

feature -- Generation

        generate
                -- Generate code for this pattern.
            require
                pattern_info_not_void: pattern_info /= Void
                pattern_info_complete: pattern_info.is_complete

invariant

        project_directory_not_empty_and_exists_if_not_void:
            project_directory /= Void implies (not project_directory.is_empty and
                                               directory_exists (project_directory))

end
```

The code generation relies on skeleton files delivered with the wizard. They are Eiffel or Ace files with placeholders of the form <SOMETHING_TO_ADD_HERE>. Here is the example of the skeleton Eiffel file that serves to generate the deferred component class of the *Decorator* pattern:

Skeleton file to generate the component class of the Decorator pattern

```
deferred class
        <DECORATOR_COMPONENT_CLASS_NAME>

feature -- Basic Operation

        <DECORATOR_FEATURE_NAME> is
                -- Do something.
            deferred
            end

end
```

The correspondence between placeholders and actual names (class names, feature names, etc.) to be generated depending on the pattern is kept in the class *PW_SKELETON_NAMES*.

To come back to the class *PW_PATTERN_CODE_GENERATOR*, its feature *generate_code* is implemented as follows:

Implementation of feature 'generate' of class PW_PATTERN_CODE_GENERATOR

```
deferred class

        PW_PATTERN_CODE_GENERATOR
...
feature -- Generation

        generate is
                        -- Generate code for this pattern.
                require
                        pattern_info_not_void: pattern_info /= Void
                        pattern_info_complete: pattern_info.is_complete
                do
                        if root_class_and_ace_file_generation then
                                generate_ace_file
                                generate_root_class
                        end
                        generate_pattern_code
                end
...
end
```

The procedures *generate_ace_file*, *generate_root_class*, and *generate_pattern_code* are deferred in class *PW_PATTERN_CODE_GENERATOR* and effected in the descendant classes. The actual implementation of these features relies on one routine *generate_code* defined in the parent class *PW_PATTERN_CODE_GENERATOR*. The signature of this feature is the following:

Signature of 'generate_code'

```
generate_code (a_new_file_name, a_skeleton_file_name: STRING;
                some_changes: LINKED_LIST [TUPLE [STRING, STRING]])
```

- *a_new_file_name* corresponds to the ".e" or ".ace" file to be generated. To use this example of the *Decorator* pattern again, if the user wants to call the deferred component class *MY_COMPONENT*, the value of *a_new_file_name* will be "chosen_project_directory_path\my_component.e" (where "chosen_project_directory_path" corresponds to the path to the project directory chosen by the user).

- *a_skeleton_file_name* corresponds to the ".e" or ".ace" skeleton file delivered with the Pattern Wizard that is used to generate the text of the new file to create (corresponding to file name *a_new_file_name*). For example, to generate the deferred component class of the *Decorator* pattern, we would use the file name of the skeleton Eiffel file given on the previous page.

See Skeleton file to generate the component class of the Decorator pattern.

- *some_changes* corresponds to the mapping between placeholders (found in the skeleton file) and the actual text to be generated. To use the example of a class *MY_COMPONENT*, the list *some_changes* would contain the tuple ["<*DECORATOR_COMPONENT_CLASS_NAME*>", "*MY_COMPONENT*"].

The actual implementation of feature *generate_code* is given below:

Full text of feature 'generate_code'

```
deferred class

        PW_PATTERN_CODE_GENERATOR

...

feature {NONE} -- Implementation (Code generation)

        generate_code (a_new_file_name, a_skeleton_file_name: STRING;
                some_changes: LINKED_LIST [TUPLE [STRING, STRING]]) is
                        -- Generate new file with file name a_new_file_name from the
                        -- skeleton corresponding to a_skeleton_file_name by
                        -- reproducing the skeleton code into the new file after
                        -- some_changes (replacing a value by another).
                        --| some_changes should be of the form:
                        --| LINKED_LIST [[old_string, new_string], ...]
                require
                        a_new_file_name_not_void: a_new_file_name /= Void
                        a_new_file_name_not_empty: not a_new_file_name.is_empty
                        a_skeleton_file_name_not_void: a_skeleton_file_name /= Void
                        a_skeleton_file_name_not_empty: not a_skeleton_file_name.is_empty
                        a_skeleton_file_exists: file_exists (a_skeleton_file_name)
                        some_changes_not_void: some_changes /= Void
                        no_void_change: not some_changes.has (Void)
                        -- no_void_old_string: forall c in some_changes, c.item (1) /= Void
                        -- no_void_new_string: forall c in some_changes, c.item (2) /= Void
                local
                        file: PLAIN_TEXT_FILE
                        skeleton_file: PLAIN_TEXT_FILE
                        text: STRING
                        a_change: TUPLE [STRING, STRING]
                        old_string: STRING
                        new_string: STRING
                do
                        create skeleton_file.make_open_read (a_skeleton_file_name)
                        skeleton_file.read_stream (skeleton_file.count)
                        text := skeleton_file.last_string
                        from some_changes.start until some_changes.after loop
                                a_change := some_changes.item
                                old_string ?= a_change.item (1)
                                if old_string /= Void then
                                        new_string ?= a_change.item (2)
                                        if new_string /= Void then
                                                text.replace_substring_all (old_string, new_string)
                                        end
                                end
                                some_changes.forth
                        end
                        create file.make_create_read_write (a_new_file_name)
                        file.put_string (text)
                        file.close
                        skeleton_file.close
                end

...

end
```

Limitations

The limitations of the Pattern Wizard are of two kinds: first, limitations of the current implementation of the tool, which should disappear in the future; second, limitations of the approach itself, which are basically the same as the limitations of this Ph.D. thesis work.

See chapter 22, page 343.

Future works on the tool include:

- Give the user the possibility to choose the root class name and creation procedure name like for the other classes.
- Give the user the possibility to use existing files (rather than always generating new files) and add to them the wished functionalities (typically adding a set of features to an existing class rather than generating a new class file with these features).

  The implied GUI changes are minor; it would suffice to add an horizontal box with a text field and a "Browse..." button to let the user choose the file to modify (in the same spirit as the project location selection).

  The major changes would be in the code generation part. It would require parsing the existing class to get an abstract syntax tree (AST) and insert into this AST the nodes corresponding to the extra code to be added, and write the augmented AST into a file.

Other limitations of the Pattern Wizard include:

- The *language specificity*: The wizard is entirely written in Eiffel and generates Eiffel files only. However, it would be quite easy to make it generate files in Java or C# for example; we would need skeleton files in those languages and maybe one or two adaptations in the wizard's code.
- The *limited number of supported patterns*: The wizard only targets five patterns (plus a few variants); these are the five non-componentizable design patterns of [Gamma 1995] for which it is possible to generate skeleton classes. However, it would be easy to extend the wizard to support more patterns; here are the required steps:

See "Design pattern componentizability classification (filled)", page 90.

  - On the model side: we would need to write the corresponding descendant of *PW_PATTERN_INFORMATION*.
  - On the code generation side: we would need to write a descendant of *PW_PATTERN_CODE_GENERATOR*.
  - On the GUI side: we would need to write the corresponding descendant of *PW_PATTERN_VERTICAL_BOX* and *PW_PATTERN_PROPERTY_SELECTOR*.
  - Finally, we would need to make the connection between the existing implementation and the new classes by extending the features *select_pattern* and *generate_code* of class *PW_INITIAL_WINDOW*, and build the *new_pattern_info* in class *PW_INITIAL_WINDOW_IMP*.

The Pattern Wizard has been designed with extensibility in mind and could be easily adapted to a broader componentization approach that would target more design patterns and more programming languages.

21.4 RELATED WORK

One of the authors of *Design Patterns*, John Vlissides, collaborated with Frank Budinsky, Marilyn Finnie, and Patsy Yu from the Toronto Software Laboratory to build a tool that also generates code from design patterns.

[Budinsky 1996].

However, this tool is different from the Pattern Wizard in many respects: first, it uses an HTML browser and Perl scripts instead of a pure object-oriented design and implementation in Eiffel; second, it generates C++ code instead of Eiffel code. The goals of the authors were to build a tool allowing a fast turn-around: they discarded other approaches using traditional programming languages as too slow (in terms of development) and not flexible enough. The tool has a three-parts architecture: the users interact with a browser (called "Presenter") written in HTML; it transmits the user input as Perl scripts to a Perl interpreter (called "Mapper"); the Perl scripts invoke a COGENT (COde GENeration Template) interpreter, which serves as code generator. They developed the COGENT interpreter for this tool.

The "Presenter" part has some commonalities with the Pattern Wizard:

- It has an intent and a motivation page providing information to the user. These elements of information are available as HTML pages with hyperlinks. (These pages give access to the chapters of *Design Patterns* in HTML format.)
- It gives users the possibility to select different generation options:
  - Users must select the names of the classes involved in a design pattern like in the Pattern Wizard. (One thing that is possible with this tool but not yet possible with the Pattern Wizard is to use existing client classes.)
  - Users may choose different options to generate different implementation versions of the same pattern; for example, a version of the *Composite* pattern favoring transparency and another one favoring safety.
  - Users may choose different code generation options; for example, they can decide to generate a main method and debug information.

See "Composite pattern", 10.1, page 147 for a detailed description of the Composite pattern and its different flavors.

The Pattern Wizard could benefit from some ideas of the "Presenter" part of the code generation tool by Budinsky et al. (for example, more fine-grained code generation options, a Questions & Answers page, etc.) to become even more user-friendly. As for the other facets (like design and architecture), the Pattern Wizard brings a new and simpler solution based on fully object-oriented design and implementation using Eiffel. As far as I know, no such tool was available for Eiffel.

21.5 CHAPTER SUMMARY

- The Pattern Wizard is a graphical application that enables generating skeleton classes automatically for some non-componentizable patterns.
- The code generation relies on template files delivered with the Pattern Wizard, which are filled according to the input given by the user. The user can also rely on default values, in which case he just has to click a button "Generate" to launch the code generation.

[Arnout-Web].

- The Pattern Wizard has been designed with extensibility in mind and could easily be extended to support other patterns and even other programming languages.
- Componentization and tool support complement each other very well.