Integrating Code Generation and Refactoring

Marcelo d'Amorim[1], Clóvis Nogueira[1], Gustavo Santos[1], Adeline Souza[1], and Paulo Borba[1,2]

[1] Qualiti Software Processes. Avenida Marquês de Olinda, 126, 4o. andar. Bairro do Recife, Recife-PE, Brazil. 50030-901

[2] Federal University of Pernambuco, Informatics Center. POBOX 7851, Recife-PE, Brazil. 50640-970

Abstract. Coder is a tool for generation and maintenance of Java programs. It is capable of not only producing class structures but also their implementation. It also allows one to safely modify the new code by applying refactorings. In this way the system evolves with reduced defects, and the productivity of the development team increases since the tool seamlessly supports coding. This position paper describes the Coder architecture, and presents an example of its practical use on projects based on Java technology.

1 Introduction

In this paper we describe a tool named Coder that integrates code generation and refefactoring in a uniform and highly flexible way. This tool can then be used for generation and maintenance of Java programs. It helps to evolve systems with a reduced number of defects at the same time that increases the productivity of the development team by seamlessly supporting different coding activities.

Our approach is to consider program transformation as a unifying concept for code generation and refactoring. A refactoring comprises several behavior preserving changes on the program, but does not add new functionalities [6]. A generator, on the other hand, introduces new functionalities. Such unifying view of transformation may create new types and modify old ones as long as it preserves the semantics of the original program. Hence, the unification integrates two essential elements to enable systematic software evolution: modification and generation of programs.

The user instruments the stated transformations and so we consider them an abstraction to *generative programming*. However they are not visible to the programmer using the tool for supporting coding activities. By using the tool the programmer manipulates only wizards, which encapsulate transformations and the associated graphical interfaces used to configure these transformations.

Coder wizards are defined by assembling transformations and graphical components reused or constructed by a tool customizer, which is also responsible for defining the wizards. The components are Java Beans that comply to a specific interface. Transformations are written in a language that extends

Java with metaprogramming constructs [1]. Such extension allows the tool customizer to program generic templates, and by using them he specifies how Java code should be generated or modified. For example, the customizer can specify that new methods should be added to an existent type, and that the protected modifiers should be replaced by private ones.

The flexibility for creating and composing wizards based on an open architecture contrasts with typical refactoring tools, which support only a fixed set of built-in refactorings. Moreover, these tools typically focus on the abstractions of a given language, organizing refactorings into method, attribute, and type categories. They do very well for generic problems like changing class packages or the name of a public method. However they do not embody application domain to deal with larger and more specific tasks. They are not able, for example, to generate or modify a set of classes according to a layered architecture. Coder wizards enable users to specify and reproduce families of components closely related to the architecture in place.

This work is outlined in the following manner: the next section describes the tool architecture. Section 3 shows its application to support Enterprise Java Beans within a layered architecture. This specific customization of the tool does not generate EJB classes and their deployment descriptors. We assume programmers can do it well by means of any EJB tool available in the market. We demonstrate how these standardized EJB types can be introduced into the system with minimal impact. Finally, section 4 presents limitations, future improvements, and concludes the paper.

2 Architecture

The design of Coder follows the layered architectural pattern [2]. Three layers are used. The first houses graphical components. The second one acts on the communication and mediation between graphical components and those of the bottom layer, which focus on program transformation.

In addition to the vertical partitioning, the presentation and mediation layers are also partitioned horizontally since their components are in charg of conceptually distinct tasks. Figure 1 depicts the dependencies between each architecture module.

The numbers on the diagram expose dependencies and also the sequence of execution of a wizard. The first dependency reveals the use of an object of type `Session` during execution. A generator quite often requires parameterization and this data can be provided through the wizard user interface. These graphical components use the session to retrieve and update wizard parameters that correspond to their fields. The session acts as a shared storage associating values to parameters that will ultimately be used to configure transformations, as sketched in the dependency 4 arrow. For instance, suppose we have a transformation that refactors some class. We need to inform

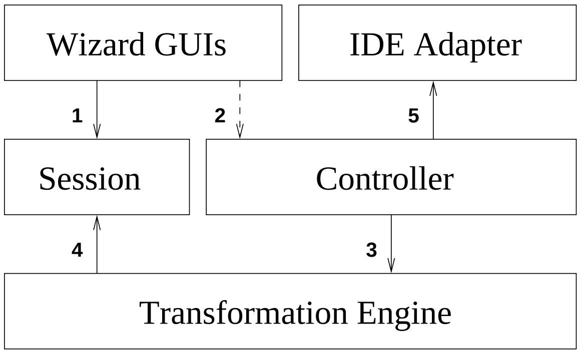

Fig. 1. Architecture layers

the name of this class to the wizard so that it can be modified later. The wizards user interfaces capture this kind of information.

As stated before wizards are defined by assembling transformations and graphical components. Actually, we can associate many graphical components to a single transformation and many transformations to a wizard. This arrangement enables modular composition of wizards from already built and tested transformations. Transformations are configured and guided by their user interfaces through the command of a programmer. During the transition between the GUIs of two transformations, the controller is notified, by means of an event, about which transformation needs to be applied—dependency 2. The controller then calls the transformation engine. The result is a set of types, created or modified, which are based on informed types.

After the last GUI transition of a wizard, the controller has also to update the IDE with all generated and modified types—dependency 5. The programmer, therefore, only perceives the results after successfully finishing all transformation steps. This decision allows him to cancel the execution without leaving the workspace inconsistent. It also aims at maintaining the IDE consistent even at the presence of unexpected situations. For example, during transformations, the power could be turned off or some configuration file could be casually removed.

For updating the programmer's IDE project, the controller uses an adapter that provides a common interface to manage programming elements of a given IDE. For each IDE the Coder integrates, there is an adapter which uses the API of that environment.

2.1 The Transformation Engine

Some wizards just generate code; others only refactor code. There are also some that require both capabilities. To support them in a uniform manner, the transformation engine realizes two operations:

- **Pattern Matching**: To make room for refactoring the tool has to manipulate types already created. This is also useful for code generation that depends on information specified by existing types. We need to represent these types even before they exist. This is accomplished by describing these types as templates. During the wizard application a template is matched with one user-informed type[1]. It is so regarded as a *source template*. The transformation engine knows and manipulates templates, not the specified types.
- **Code Generation and Update**: Similarly to C++ which allows parametric polymorphism[2] as a means to extend the definition and use of language abstractions, it is possible to define templates for types that will be generated or modified. These templates should safely generate code to the target language. We regard these as *target templates* because they actually modify code. A target template is allowed to manipulate source templates as if they were actual types, in which it depends on.

There is not an one-to-one relation between source and target templates. We can define a source template specifying type T without a target. The inverse is also possible, and it is also possible to have both kinds of templates for the same type. In the first case the actual type represented by T is only analysed. In the second case, a new type is generated, and in the last situation the matched type is modified. This is the case of refactoring. Figure 2 sketches this situation. ST stands for source template and TT for target template.

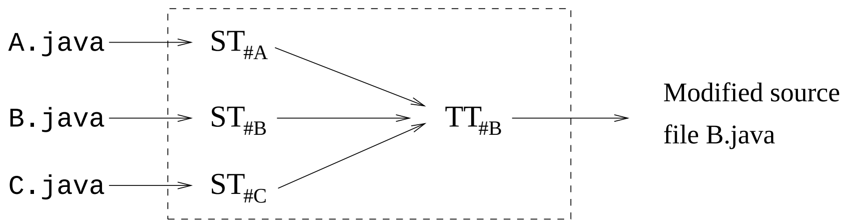

Fig. 2. A set of templates for refactoring

Suppose we want to modify the code of an user-defined class `B`, and such modification depends on the information provided by two other types, say interfaces `A` and `C`. The result of the update, for example, provides `B` with an empty implementation for the methods in `A` and `C`. This is actually the

[1] This user-directed matching differs from dynamic pattern matching which is very useful in dependency analysis of code. This subject is further discussed on the Conclusions.

[2] C++ is an example of language that supports the definition of generic classes and functions. The work of Myers et. al [5] proposes the introduction of these concepts to Java.

Inherit[**Base, Derived**] refactoring as described in [7]. This kind of transformation is done quite directly by the engine. The target template $TT_{\#\mathrm{B}}$ uses metaprogramming operators to instrument the informed interfaces and then include empty implementations for their methods. In fact, target templates are written in the JaTS language, an extension of Java with some metaprogramming operators. The templates mentioned on Figure 2 rely on these operators. Concrete examples are provided on the Applications section. Transformations are made on the syntax tree of the original parsed programs. This assures that the resulting programs are syntactically valid if the transformations are valid.

2.2 Wizards

Wizards are units of code transformation manipulated by programmers. A wizard is composed of a set of transformations applied in sequence, and a set of GUIs presented prior to each transformation. Graphical interfaces serve to configure the transformations. However, there might be transformations without associated GUIs. A parameterless code generator is an example. Considering the symbol g represents a GUI and t a transformation, the following sequence describes a wizard with four transformation steps:

$g_1\ g_2\ g_3\ t_1\ g_4\ g_5\ g_6\ t_2\ g_7\ g_8\ g_9\ t_3\ t_4$

Each transformation is applied immediately after the programmer informs the configuration data for that transformation. For example, the transformation t_1 is applied during the transition from g_3 to g_4. On the other hand, results are only presented to the programmer at the end of the wizard execution. The following grammar provides a syntactic representation for wizards:

$$\begin{aligned} W &\rightarrow GTL \\ GTL &\rightarrow GT \mid GT, GTL \\ GT &\rightarrow (GL, T) \\ GL &\rightarrow \epsilon \mid g, GL \\ T &\rightarrow (st^{\star}, tt^{+}) \end{aligned}$$

where GTL is a non-empty list of pairs. The first element of the pair is a list of GUIs (GL), and the second a transformation (T), which can include many source templates (st), but at least one target template (tt). The pair GT is a unit of transformation that can be used to define other wizards in a modular way.

This grammar is defined as an XML-Schema. It is used to validate any XML document describing an wizard (the wizard descriptor). This document is in strict accordance to the grammar described above, thus it lists templates and GUIs used on transformations. A compressed file includes, in addition to the wizard descriptor and its associated grammar, graphical interfaces and templates the wizard requires. This file bundles the wizard in a single easily deployable unit.

2.3 GUI Coordination

The programmer should be capable of navigating through the GUIs of a wizard, including returning to an edited screen and correcting the value of some field. The programmer, user of the tool, does not manipulate transformations directly. Hence, the sequence of GUIs followed by transformation does not reflect his vision. The user observes only a sequence of GUIs, which can be navigated at any direction.

Rather than including transformations into the wizard's sequence of execution, like presented before, we include event interceptors. This allows to asynchronously communicate an event in a special transition between two screens and thus to reduce coupling between the controller and graphical components. The controller does not need to manage the sequence of screens to be shown, this is done in a distributed manner by the graphical components. The following sequence is an operational equivalent version of the sequence (1). The symbol i represents an interceptor:

$$g_1\ g_2\ g_3\ i_4\ g_5\ g_6\ g_7\ i_8\ g_9\ g_{10}\ g_{11}\ i_{12}\ i_{13}$$

Each element of the sequence is identified by an index. At the beginning of the wizard execution, an one-valued token is created. As the wizard screen sequence proceeds or recedes, the value is updated and a parameterized event with that value is thrown. Only the element with index equal to the token value should catch and process the event. When the element is an event interceptor, it should notify the controller to apply the proper transformation and increment the token in the following.

This is a variation of the Phased Process pattern [8], requiring no dispatcher component to guide the execution process.

3 Applications

Within the object-oriented layered architectural pattern, persistency is an aspect implemented by the data layer. Data collections are types that deal with this aspect [4]. These components implement an interface known as business-data interface, which regulates the interaction between data and business elements. The structure of a data collection can vary considerably according to the persistent mechanism and technology being used.

The data collection gathers access and maintenance methods for a given entity type of the system. On the other hand, the EJB specification spreads these operations over distinct interfaces: the Remote interface of an Entity Bean is responsible for updating the entity data, and the Home interface is responsible for locating and removing the entity.

The Adapter design pattern can resolve these differences. We can use an adapter to implement the business-data interface by means of the two EJB interfaces. These interfaces are not accessible to the application business objects, but only to the adapter. Hence, the decision about which technology to use in this layer does not affect business rules. Such structure is detached in the diagram of Figure 3.

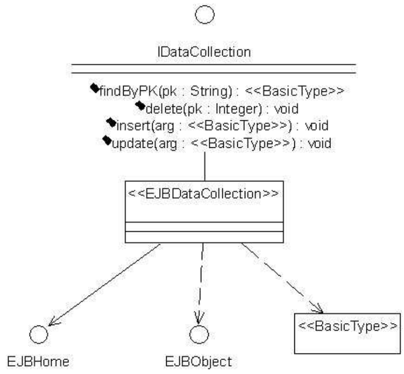

Fig. 3. EJB Data Collection

For a given entity (business basic type), the **EJB Data Collection** wizard generates code according to the mentioned architectural pattern. As stated in [9] we believe that refactorings have a strong connection with patterns:

> *Many refactorings are about introducing patterns into a system. There is a natural relation between patterns and refactorings. Patterns are where you want to be; refactorings are ways to get there from somewhere else.*

Refactoring can be used as a technique to automate pattern implementation, and so does code generation. The illustrated wizard uses only code generation, but a similar one could do some refactoring too.

The EJB Data Collection wizard can be used as many times as the number of persistent entities in the system. While applying this wizard, it is possible to disregard multiplicity and the direction of associations among business entities. As a consequence, there is no order to apply the wizard. Figure 4 illustrates the mentioned wizard under JBuilder, where one can see a window for collecting information necessary for generating code.

Another wizard complements this one in order to support the stated architectural pattern generation. It includes multiplicities into data collections, by refactoring the code generated by the previous wizard. A refactoring of this type affects both the business basic type (entity) and its data collection. We call this refactoring **Dependency Inclusion**. In the following we

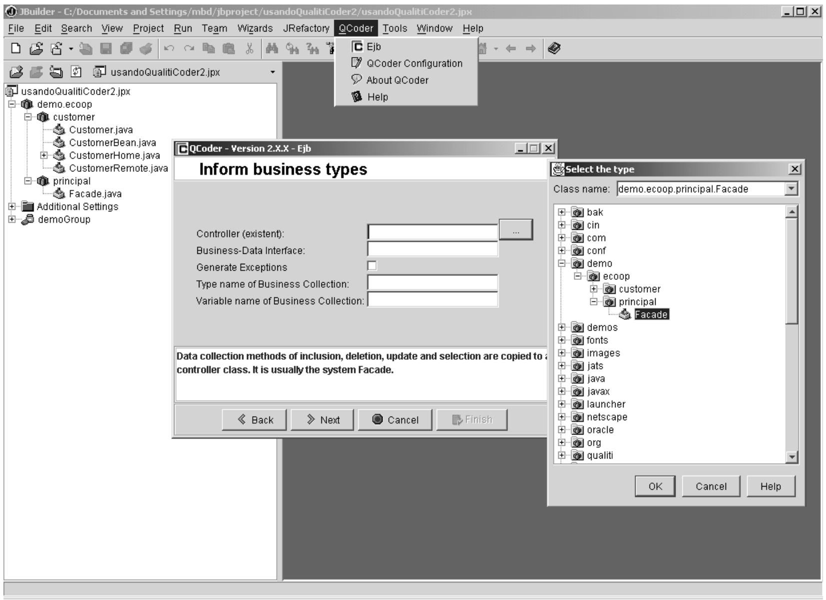

Fig. 4. Coder under JBuilder6.0

present a code fragment of the `Customer` data collection after including a 1–1 dependency between `Customer` and `Account`:

```
1: public Customer findByPK(String customerID) ... {
2:
3:    CustomerRemote customer_remote = null;
4:    Customer customer = null;
5:    try {
6:        customer_remote = home.findByPrimaryKey(customerID);
7:        customer = customer_remote.getValueObject();
8:        // ******* built on refactoring
9:        String accountID = null;
10:       Account account = null;
11:       AccountRemote account_remote = null;
12:       accountID = customer_remote.getAccountID();
13:       account_remote = account_home.findByPrimaryKey(accountID);
14:       account = account_remote.getValueObject();
15:       customer.setAccount(account);
16:   }
17:   catch (...) {...}
18:   return customer;
19: }
```

At line 6, the method `findByPrimaryKey` returns a reference to the Entity Bean remote interface, which is then used to retrieve a remote version of the object. The remote reference is stored into the `customer` variable. The commands between lines 9 and 15 are the result of the refactoring which includes a dependency to an object of type `Account`. In line 12, the customer account identification is retrieved. The Home and Remote interfaces are then accessed to retrieve an `Account` object. This object is finally linked to the `Customer` at line 15.

Source and target templates have been defined to accomplish these modifications. The modification on the `findByPK` method was guided by a target template on the data collection. Unfortunately it is a lengthy template to show here. However, some modification is also required on the entity type, so there are templates on those types too. Here we show a short example of such templates, which were used to refactor the `Customer` class within this wizard. They include the entity class an attribute to the dependent type.

We have to provide a source template in order to match the informed type so that it can be further instrumented:

```
<< #PD:PackageDeclaration; >>
#IDS:ImportDeclarationSet;

#M:ModifierList class #BASIC_TYPE <<extends  #BC_BASIC_TYPE >>
        <<implements  #IFS_BASIC_TYPE>>{
    #ATTRS:FieldDeclarationSet;
    #MTDS:MethodDeclarationSet;
}
```

`#BASIC_TYPE` is an identifier representing the name of the type to be matched. Using the tool the programmer indicates that `Customer` is the type to be matched with this template. After a successful matching the identifiers store valuable information which can be used to instrument transformations. The engine matches these identifiers automatically.

The following target template is based on the same type of the source template. Note that `#BASIC_TYPE` appears as the name of the class in both templates. When the target template is processed the actual type to which `#BASIC_TYPE` maps was already matched, and this type name will be used as the name of the resulting class.

```
<< #PD:PackageDeclaration; >>
#IDS:ImportDeclarationSet;

#M:ModifierList class #BASIC_TYPE <<extends  #BC_BASIC_TYPE >>
        <<implements  #IFS_BASIC_TYPE>>{
    #ATTRS:FieldDeclarationSet;
    #DEPENDENCY #DEPENDENCY_NAME;
    #MTDS:MethodDeclarationSet;

    public void [[ #X.addPrefix("set") ]] (
      #DEPENDENCY obj) {
        #DEPENDENCY_NAME = obj;
```

```
    }
}
```

Note that some identifiers were not declared in the source template. For example, the `#DEPENDENCY` identifier was matched through the source template of the dependent type. This means that there is a similar source template (not shown) for the dependent type and the programmer informed the tool which type should be matched with it. The `#DEPENDENCY_NAME` identifier, however, is not automatically matched like the others. It is directly informed via the wizard GUI along with the `Customer` and `Account` types.

In order to realize algorithmic transformations on Abstract Syntax Trees we use executable expressions on target templates. They are represented with double brackets and are processed after the programmer has provided all information to the wizard. There is a set of operations allowed for each kind of node. In the example above, we use the Java `addPrefix` method on a `Type` node in order to define a method name. The expression gives `setAccount` as result.

4 Conclusions

We described in this paper the architecture of a tool to support code transformation in a very configurable way. We have also illustrated applications of this tool for both code generation and refactoring, showing how it can be customized to deal with an specific EJB pattern.

A previous version of Coder is currently in professional use. This version is very specialized on the generation of code according to an architectural layered pattern. It allows modification of templates but do not provide facilities to easily include new generators without changing the tool. The transformation engine is also very limited in the first version. There are neither pattern matching nor metaprogramming operators.

The next release of Coder is being delivered in June 2002 to a software department that uses EJB technology to develop Web applications. The tool will be configured with 2 wizards: EJB Data Collection and Dependency Inclusion, which have been mentioned in this paper.

We argue that the major aim of the tool is the facility to build new wizards. Templates are written in a language that extends Java with metprogramming constructs. Thus we believe Java programmers will probably have a reduced learning curve. In addition, wizards GUIs can be developed with higher degree of freedom. They barely communicate with the framework. They just use a shared storage to provide and retrieve parameters. All these pieces are tied together in a declarative manner by a wizard descriptor file, and bundled together into a single and easy-to-deploy jar file.

We believe that other forms of matching are an interesting matter to investigate. There are refactorings which modify a variable number of classes. This is quite usual and the Coder still does not support it. The modification of a public method name is a known example. Rather than asking the user to inform the class that will match a template the tool should discover by analyzing dependencies which class match that template. A source template could then match a set of dependent classes and a target template in turn modifies the matched set by means of metaprogramming operators.

As another limitation, Coder still does not support close integration with other languages. We need, for example, to generate HTML and Java Server Pages based on some public operation. These pages serve to the input or presentation of object's data. If we need to store a `Customer` through the Web we have to provide HTML fields to input the `Customer` and `Account` fields.

References

1. Fernando Castor and Paulo Borba. A language for specifying Java transformations. In V Brazilian Symposium on Programming Languages, pages 236-251, Curitiba, Brazil, 23rd-25th May 2001.
2. Felix Bachmann et al. Software Architecture Documentation in Practice: Documenting Architectural Layers, CMU Technical Report CMU/SEI-2000-SR-004, March 2000.
3. Krzysztof Czarnecki, Ulrich W. Eisenecker. Generative Programming - Methods, Tools, and Applications. Addison-Wesley, June 2000.
4. Tiago Massoni, Vander Alves, Sérgio Soares, and Paulo Borba. PDC: The persistent data collections pattern. In First Latin American Conference on Pattern Languages of Programming, Rio de Janeiro, Brazil, 3th-5th October 2001.
5. A. C. Meyers, J. A. Bank, and B. Liskov. Parameterized types for Java. In Symposium on Principles of Programming Languages, pages 132–145. ACM, 1997.
6. William F. Opdyke. Refactoring object-oriented frameworks. PhD thesis. Urbana-Champaign, IL, USA, 1992.
7. Lance Tokuda and Don Batory. Automating Three Modes of Evolution for Object-Oriented Software Architecture. Proceedings 5th USENIX COOTS'99. San Diego, California, USA. 3th–7th May, 1999.
8. Jon Benton. Evolve your apps with the Phased Process pattern. Java World, http://www.javaworld.com. April 5, 2002.
9. Martin Fowler et al. Refactoring: Improving the Design of Existing Code. Addison Wesley. November 1999.