IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP5001

Integrating Design Patterns into Forward Engineering Processes

Liliana Martinez
INTIA, Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina, lmartine@exa.unicen.edu.ar

Liliana Favre
INTIA, Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina, CIC (Comisión de Investigaciones Científicas de la Provincia de Buenos Aires), lfavre@exa.unicen.edu.ar

Claudia Pereira
INTIA, Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina, cpereira@exa.unicen.edu.ar

ABSTRACT

Design patterns are reusable solutions to recurring problems that occur during software development. Most UML CASE tools do not assist in the integration and code generation of design patterns, this task is still left to the programmer. This paper describes a forward engineering process of UML static models which supports working with design patterns. This approach is based on the integration of semi-formal notations in UML with algebraic specifications. Transformations are supported by a library of reusable schemes and by a system of transformation rules which allow translating UML to algebraic specifications and object oriented code step by step. All the proposed transformations could be automated; they allow traceability and could be integrated into iterative and incremental software development process. Eiffel was chosen to show the feasibility of our approach.

1. INTRODUCTION

Design patterns are a useful concept to guide and to document the object-oriented software system design. Their contribution covers definition, design and documentation of class libraries and frameworks. Even if there is no consensus about the way to support the application of design patterns (using tools, languages, etc) in the software development, this task should be automated or at least assisted. The industrial experience (Beck et al., 1996) indicates that design patterns increase the speed of the development of systems and facilitate the communication, although they are hard to write. Manual application is tedious and error prone. The loss of traceability is a drawback of the by-hand-coding task (Albin-Amiot & Guéhéneuc, 2001a). When a pattern is applied, the resulting implementation does not provide a means to go back to the pattern from which it was derived, the pattern code being mixed within the user application code.

Design patterns have been widely accepted by software practitioners. Several IDE (Integrated Development Environments) and UML modeling software environments have begun to introduce support for the design patterns (OMG, 2003). Most CASE tools generate code from UML design. Few UML CASE tools provide assistance to the programmer in the integration of code automatically generated for the patterns in their applications.

On the other hand, most existing pattern tools simply assist in "cut and paste" processes, whereby the designer selects a pattern and obtains a code piece in the appropriate language to incorporate it into the implementation. The techniques are not, in general, independent of the language and they are unable to generate code in more than one language. The programmer needs then to adjust the code to the implementation (Peckman & Lloyd, 2003).

These approaches assume that the patterns involve classes dedicated to their role as collaborator within a particular design pattern. This is not true in general: the patterns rarely exist in isolation. It frequently happens that a *collaborator* in one pattern plays a role in another. The pattern definition emphasizes this fact: a pattern is a solution to a problem in a particular context. Also, a pattern is implemented producing not only new classes and routines, but more often adapting the existent context, i.e., preexistent program constructions, to the roles they assume in the newly applied pattern. A more beneficial application of a design pattern should adapt the existent construct in a program, a task more complex and context-dependent than the code generation by hand (Eden et al., 1997).

Bulka (2002) analyzes state of the pattern automation tools and discusses the pros and cons of several approaches. It establishes that there are several degrees of pattern automation offered by the UML modeling tools. They go from simple template to intelligent patterns. The simple template approach simply inserts a group of related classes in a workspace (e.g. UMLStudio) The intelligent pattern approach attempts to integrate classes from the newly inserted pattern with the existing classes, renaming classes and methods as required and responding to changes in other parts of the UML model (e.g. ModelMaker).

Favre et al. (2003) describe a rigorous process to transform UML static models into object oriented code. A transformational approach is introduced for the integration of the UML static diagrams with algebraic languages and object oriented code. Our current contribution is towards an embedding of the detection and code generation of design patterns in a rigorous process which facilitates reuse and evolution. This process is based on the combination of UML semiformal notations and algebraic specifications and it is guided by rules to translate step-by-step UML constructions allowing traceability. The NEREUS language is used to describe the static structure of a design pattern (Favre, 2003) Eiffel was the language of choice to show the feasibility of our approach.

This paper is organized as follows. Section 2, deals with the related work. Section 3 outlines the NEREUS language. Section 4 describes the basis of a forward engineering method. Finally, Section 5 considers conclusions and future work.

2. RELATED WORK

In Budinsky et al. (1996) a tool to automatically generate code of design patterns from a small amount of information given by the user is described. This approach has two widespread problems. Invasion, the user should understand "what to cut" and "where to paste" and both cannot be obvious. It is not reversible, once the user has incorporated pattern code in his application, any change that implies to generate the code again will force it to reinstate the pattern code in the application. The user cannot see changes in the generated code through the tool.

Florijn et al. (1997) describe a tool prototype that supports design pattern during the development or maintenance of object-oriented programs.

Albin-Amiot & Guéhéneuc (2001a) describe how a metamodel can be used to obtain a representation of design patterns and how this representation allows both automatic generation and detection of design patterns. The wanted contribution of this proposal is the definition of design patterns as entities of modeling of first class. The main limitation of this approach concerns the integration of the generated code with the user code.

Albin-Amiot & Guéhéneuc (2001 a) present two tools (Scriptor and PatternsBox) that help the developers to implement large applications and large frameworks using design patterns. In Scriptor the developers have little or no control on the generated code, once the code is generated, there is no form of locating what design pattern has been applied and where it has been applied. PatternsBox (preservative Generation) tool allows us to instance design patterns. The developers need to write most or great part of the code by hand.

3. THE NEREUS LANGUAGE

NEREUS (Favre, 2003) is a notation to formally specify UML static diagrams. NEREUS is based on GSBL^{oo} (Favre, 2001). As GSBL^{oo} is relation centric: it expresses different kinds of relations as primitives to develop specifications. The characteristic that distinguishes NEREUS from GSBL^{oo} is its language neutrality. NEREUS is open to many algebraic languages including CASL and Larch.

The syntax of a basic specification is presented below:

CLASS className [<parameterList>]
IMPORTS <importList>
INHERITS <inheritsList>
DEFERRED
TYPE <sortList>
FUNCTIONS <functionList>
EFFECTIVE
TYPE <sortList>
FUNCTIONS **<functionList>**
AXIOMS
    <varList>
    <axiomList>
END-CLASS

In NEREUS generic classes can be distinguished by means of explicit parameterization. The IMPORTS clause expresses dependency relations. Subclassing is expressed in the INHERITS clause, the specification of the class is built from the union of the specifications of the classes appearing in the *<inheritsList>*.

NEREUS allows us to define local instances of a class in the IMPORTS and INHERITS clauses by the syntax *className [<bindingList>]* where the elements of *<bindingList>* can be pairs of sorts *s1: s2*, and/or pairs of operations *o1:o2* with *o2* and *s2* belonging to the own part of *ClassName*.

Sorts and operations are declared in the TYPE and FUNCTIONS clauses. In NEREUS it is possible to specify any of the three levels of visibility for operations: public, protected and private. These are expressed by prefixing the symbols: +, #, and - respectively. If the operation is not decorated with a symbol of visibility, it can be assumed it is public.

NEREUS distinguishes deferred and effective parts. The DEFERRED clause declares new sorts or operations which are incompletely defined. The EFFECTIVE clause either declares new sorts or operations which are completely defined, or completes the definition of some inherited sort or operation.

NEREUS supports higher-order operations (a function *f* is higher-order if functional sorts appear in a parameter sort or the result sort of *f*). In the context of OCL Collection formalization, second-order operations are required. It is possible to limit the scope of the declarations of auxiliary symbols by using local definitions. Also, NEREUS allows us to specify incomplete signatures by using the underscore notation:

$$f : d_1 \ x \ d_2 \ x \ \_ \ \rightarrow \ r$$

NEREUS provides a taxonomy of constructor types which classifies binary associations according to: its kind (aggregation, composition, association, association class, qualified association), its degree (unary, binary), its navigability (unidirectional, bi-directional) and its connectivity (one-to-one, one-to-many, many-to-many).

Generic relations can be used in the definition of concrete relations by instantiation. New associations can be defined by the following syntax:

ASSOCIATION <relationName>
IS <constructorTypeName> [...: Class1; ...:Class2; ...:Role1; ...:Role2;
                ...:mult1; ...:mult2; ...:visibility1; ...: visibility2]
CONSTRAINED-BY <constraintList>
END

The **IS** clause expresses the instantiation of *<constructorTypeName>* with classes, roles, visibility and multiplicity. The **CONSTRAINED-BY** clause allows the specification of static constraints in first order logic.

CLASS C
ASSOCIATES
<<relationName>>
...
END-CLASS

The keyword **ASSOCIATES** identifies ordinary associations. An association may be extended to have its own set of operations and properties. Such an association is called an association class.

The package is the mechanism provided by NEREUS to group classes. It matches the UML semantics. Classes and their relations from the system design might be separated into a series of packages, using the NEREUS import dependencies to control access among these packages.

Several useful predefined types are offered in NEREUS, for example *Collection*, *Set*, *Sequence*, *Bag*, *Boolean*, *String*, *Nat* and enumerated types.

NEREUS is an intermediate notation open to many other formal languages. In particular, we define its semantics by giving a precise formal meaning to each of the construction of the NEREUS in terms of the CASL language, due to it is a unifier of proven algebraic languages (Astesiano et al., 2002).

4. FORWARD ENGINEERING THROUGH DESIGN PATTERNS

At design level, the patterns are represented through constructions of high level (such as classes, methods and relationships) in an implementation independent way. In the existing design pattern catalogs (Gamma et al, 1995; Sherman et al, 1998), the pattern description is made through text, UML diagrams and code examples of alternative implementations of the same ones.

As a rule, a pattern is expressed in a design vocabulary richer than one offered by the programming languages or design notations. A design pattern is not just a solution structure consisting of a UML diagram, it is more than classes and relationships.

We propose a forward engineering process of UML static diagrams which support working with design patterns. The UML/OCL is used to generate high level specifications in NEREUS. These specifications are tailored to specify realizations that fit a specific technology, which in turn are used to generate the code.

There is a need for reusable and adaptable components. We define specific reusable components for associations, OCL Collections and design patterns.

A component is defined in three levels of abstraction: specialization, realization and implementation. The specialization level describes

components with a high level of abstraction, which is independent of any implementation technology. This level defines a hierarchy of incomplete specifications as an acyclic graph. The specialization level has two views. One of them is based on NEREUS and the other in UML/OCL. OCL helps the user in the component identification process without forcing him to change the specification style. Specifications in the specialization level are linked with subcomponents at the realization level. The realization sub-components are trees of algebraic specifications: the root is the most abstract definition, the internal nodes correspond to different realizations of the root. The realization specifications fit a specific technology. For example, for the *Composite* component (Gamma et al., 1995 p. 163) the specialization level integrates incomplete specifications, some of them allow us to add the same element twice. Different realizations through sequences, set or bag could be associated. The implementation level associates each leaf of the realization level with different implementations that are object-oriented code pattern.

A specific reusable component is *Association.* The specialization level describes a taxonomy of associations classified according to kind, degree, navigability and multiplicity. Every leaf in this level corresponds to sub-components at the realization level. For example, for a "binary, bi-directional and many-to-many" association, different realizations through hashing, sequences, or trees could be associated. Implementation sub-components express how to implement associations and aggregations. For example, a bi-directional binary association with multiplicity "one-to-one" will be implemented as an attribute in each associated class containing a reference to the related object. On the contrary, if the association is "many-to-many", the best approach is to implement the association as a different class in which each instance represents one link and its attributes

The component reuse is based on the application of reuse operators: *Rename*, *Hide*, *Extend* and *Combine*. These operators were defined on the three levels of components (Favre et al, 2003). **4.1. Description of the steps of the method**

Figure1 shows the main steps of the method.

Starting from UML class diagrams, an incomplete algebraic specification can be built by instantiating reusable schemes and classes which already exist in the NEREUS predefined library. Analyzing OCL specifications we can derive axioms that will be included in the NEREUS specifications. Preconditions written in OCL are used to generate preconditions in NEREUS. Postconditions and invariants allow us to generate axioms in NEREUS. (Favre et al. 2000, Favre, 2001). Thus, an incomplete algebraic specification containing the highest information extracted from UML model can be built semi-automatically. The refinement of the NEREUS incomplete specification into the complete algebraic specification and code is based on a library of reusable components.

The algebraic specification is used to detect patterns into the design by means of a signature matching (the objects are known through their interfaces, i.e., through the signatures of their operations). To support this process, a library of reusable components was built. It contains schemes of design patterns in NEREUS. Gamma et al. (1995) patterns were considered.

Figure 1. From UML/OCL to code

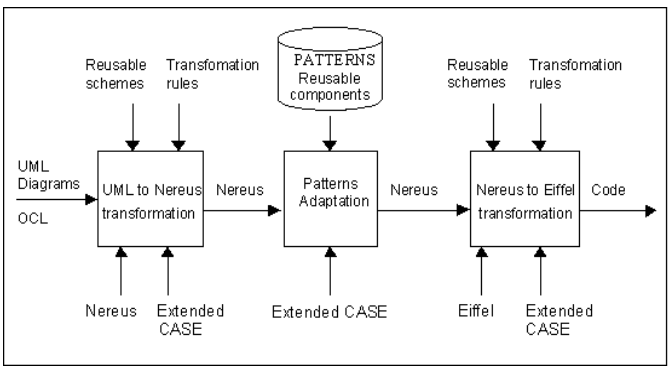
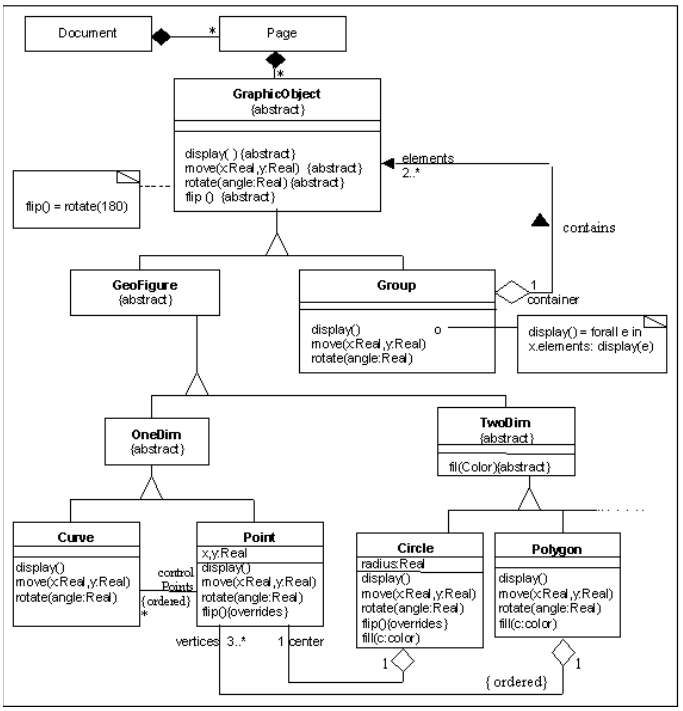


Figure 2. A diagram Editor



When a pattern is detected, its specification is integrated to the specification of the UML diagram. An implementation is selected and the code is generated and integrated with the code corresponding to the rest of the UML diagram.

To carry out this task, a library of schemes of design patterns written in Eiffel was built (for each of the patterns of the library, the programmer has different implementations). The relation introduced in NEREUS using the clause IMPORTS will be translated into a client relation in Eiffel. The relation expressed through the keyword INHERITS in NEREUS will become an inheritance relation in Eiffel. Associations are transformed by instantiating schemes that exist in the reusable component Association. For every ASSOCIATES clause, a scheme in the implementation level of the association component will be selected and instantiated.

4.2. Example

Figure 2 shows a hierarchy of classes of a diagram editor presented in (Mandel & Cengarle, 1999). The editor supports the notion of group of graphic elements. A document consists of pages and a page consists of graphic elements. Graphic elements are either geometric figures or groups of at least two graphic elements; a graphic element can be a member of a group at most. Graphic elements can be moved, rotated, etc.

The diagram in Figure 2 can be enhanced with OCL constraints that further restrict the possible system states. For example:

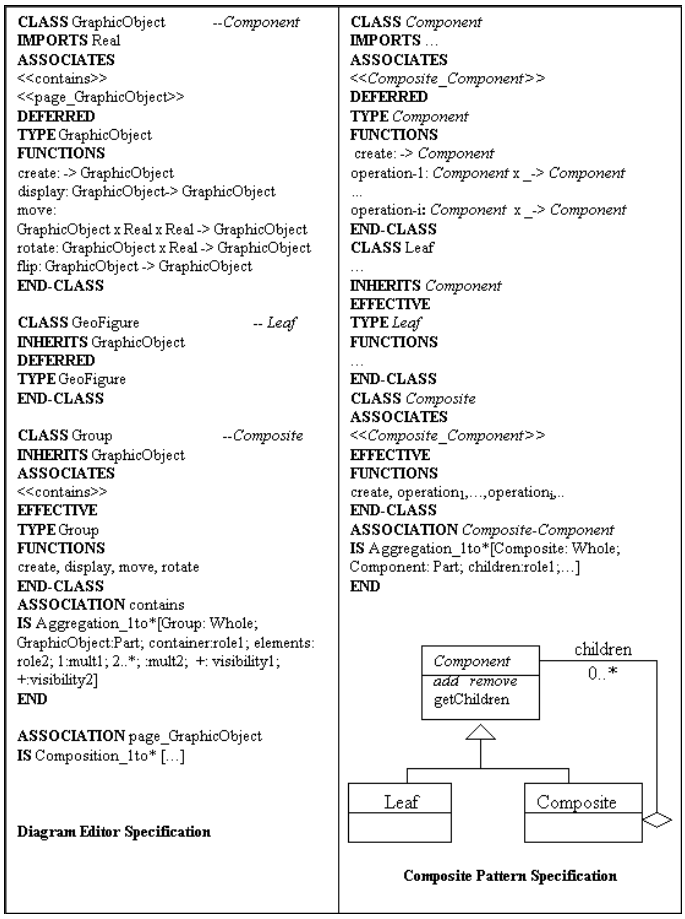
Group
self.elements->size ³ 2

Group::display (x:Real, y:Real)
post: result = self.*elements* -> iterate(*component* |*component*.display)

Figure 3 shows the partial specification in NEREUS of the UML class diagram shown in Figure 2 which corresponds to the *Composite* pattern. Also, it shows the *Composite* pattern scheme in NEREUS.

The diagram can be automatically translated into a NEREUS algebraic specification. Starting from this specification, the presence of the *Composite* pattern can be detected by means of signature matching. The detection process allows generating complete and integrated code.

Figure 3. Composite pattern identification



For the classes of the *Composite* pattern it will be generated operation signature, pre-conditions and post-conditions of operations and class invariant, but also the code of some operations.

Composite-Component aggregation is transformed by instantiating the *Aggregation_1to** scheme. This instantiation includes the *add*, r*emove* and *getChildren* operations. The number of components of the Composite is limitless, the multiplicity indicates 2..*, for this reason those are stored in a c*ollection*, in particular, a linked list was selected. On the other hand, the code corresponding to the features can be generated by reusing the *Iterator* class of the Eiffel Library (Favre et al., 2003).

5. CONCLUSIONS

This work presents a rigorous process to forward engineering UML static models, which supports working with design patterns. This approach is based on the integration of semiformal notations with algebraic techniques and it is guided by rules to translate step by step UML constructions allowing traceability.

The proposed transformations preserve the integrity between specifications and code. Most of the transformations can be undone, which provides great flexibility in code generation process supported by the existing UML CASE tools. Following this approach we can use the transformations and apply them backward to reverse engineer code to a UML diagram.

Our approach depends on the availability of a large catalog of patterns which covers different implementations. Currently, a library of reusable components linked to design patterns is under construction.

A crucial problem is how to detect sub-diagrams which can be matched with a pattern. To date, the identification of design patterns by signature matching and semantic matching is being analyzed.

We foresee the integration of our results in the existing UML CASE tools environments

REFERENCES

Albin-Amiot H. & Guéhéneuc Y. (2001a). Meta-modeling Design Patterns: application to pattern detection and code synthesis. *Proc. of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, Budapest, Hungary.

Albin-Amiot H. & Guéhéneuc Y. (2001b). Design Pattern Application: Pure-Generative Approach vs. Conservative-Generative Approach. *Proceedings of OOPSLA Workshop on Generative Programming*, Florida, USA.

Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P., Sannella, D. & Tarlecki, A. (2002). CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2), 153-196.

Beck D., Coplien J., Crocker, R., Dominick, L., Meszaros, G. & Paulisch, F. (1996). Industrial experience with design patterns. *Proc. of ICSE-18 (International Conference on Software Engineering)*, Technical University of Berlin, Germany, 103- 113

Budinsky, F., Finni, M., Vlissides, J. & Yu, P. (1996). Automatic code generation from design patterns. *IBM System Journal*, Vol 35, N° 2.

Bulka, A. (2002). Design Pattern Automation. *Third Asia-Pacific Conference on Pattern Languages of Programs (KoalaPLoP 2002)*, Melbourne, Australia.

Eden, A., Yehudai, A. & Gil, J. (1997). Precise specification and automatic application of design patterns. *1997 International Conference on Automated Software Engineering (ASE' 97)*, Lake Tahoe, Canada, 143.

Favre, L., Martinez, L. & Pereira, C. (2000). Transforming UML Static Models to Object Oriented Code. *Technology of Object-Oriented Languages and Systems, TOOLS 37, IEEE Computer Society*, 170-181.

Favre, L. (2001) A Formal Mapping between UML Static Models and Algebraic Specifications. *Lecture Notes in Informatics (p. 7) SEW Practical UML-Based Rigorous Development Methods- Countering or Integrating the eXtremists* (Eds. A. Evans, R. France, A. Moreira, B. Rumpe) GI Edition, Konner Kollen-Verlag, 113-127.

Favre, L., Martinez, L. & Pereira, C. (2003). Forward Engineering and UML: From UML Static Models to Eiffel Code. *UML and the Unified Process* (Liliana Favre editor). Chapter IX., IRM Press, USA, 199-217.

Favre, L. (2003). The *NEREUS* Language. *Technical Report. Software Technology Group*, INTIA. Universidad Nacional del Centro de la Pcia. de Buenos Aires, Argentina.

Florijn, G., Meijers, M. & van Winsen, P. (1997). Tool support for object-oriented patterns. *Proc. of ECOOP (European Conference on Object Oriented Programming) '97*, Jyväskylä, Finland, 472-795.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Mandel, L. & Cengarle M. (1999). On the Expressive Power of OCL. *FM'99 - Formal Methods: World Congress on Formal Methods in the Development of Computing Systems*, Toulouse, France, Volume I., Springer-Verlag, 854-874.

OMG (2003) *Unified Modeling Language Specification*, v. 1.5. Object Management Group. Available at www.omg.org

Peckham, J. & Lloyd S. (2003). Integrating Patterns into CASE Tools. *Practicing Software Engineering in the 21st Century*. Chapter 1, IRM PRESS, 1-8.

Sherman, R., Brown, K. & Woolf, B. (1998). *The Design Patterns Smalltalk Companion*. Addison-Wesley.