Tool Support for Design Patterns

Gabor Karsai[1]
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract

Design patterns have been widely recognized as important contributors to the success of software systems, yet there is little tool support for their application. In this paper an approach is presented that outlines how graph rewriting techniques can be used to build tool support for design patterns. The paper considers design patterns as graph rewriting rules to be applied in class diagram, and it presents an example for its application.

Introduction

Since the arrival of the design patterns (a.k.a. "GOF") book [3], software design patterns have become part of the toolbox of every software developer and practitioner. While design patterns are perhaps as old as software development itself [2], the book was the first, widely accepted and adopted, collection of patterns that semi-formally captured what patterns are and gave numerous examples for their definition and use. Since then, the patterns movement has matured [4], which keeps expanding the collection and promotes the use of patterns in the development processes. While it is hard to measure, it seems that the software industry has embraced design patterns, engineers are using them in their everyday work.

A design pattern, by definition, gives a prototypical solution to a recurring design problem in a context. When a designer faces a design problem, understands the conditions under which the pattern is applicable and the forces impacting on the design, then he can choose an appropriate pattern and apply it in the context of the application being developed. This process was shown in numerous practical examples both in the book [3] and in several other sources [4]. "Applying a pattern" means that the prototypical solution, which is usually expressed in the form of class and code fragments, is tailored and adapted to the particular, domain-specific problem that the designer is solving.

Pattern application looks like a somewhat mechanistic process, yet with the notable exception of a few research projects [8][9][10], there is very little tool support for it. Patterns are well documented in the literature, yet it seems that very little has been done to precisely specify them and/or connect them to a formal development process. A formal development process is a method of software construction, which considers the design as a mathematical artifact, starts with a precise specification of the requirements, and through incremental, correctness-preserving transformations and extensions arrives at a detailed design, which can be automatically compiled into an executable implementation. While formal development processes are obviously "heavy-weight," they carry the benefit of being able to provide "automatic" verification of the product.

Even if a formal process is not used, being able to document design patterns in a structured form, and having tool support for their application in a development environment could lead to significant increases in productivity. One can imagine the benefits of a development tool that allows a designer to construct a system by continuously growing and weaving together classes, design patterns, and other ingredients, and which also checks and verifies the composition (along the lines of the methodology implied in the Specware environment [11]).

To build tool support for design patterns, one has to answer (at least) three questions:

1. How to represent design patterns in a structured form?
2. How to facilitate the application of design patterns thus represented?
3. How does the compositional design process work in this context?

[1] gabor@vuse.vanderbilt.edu

This paper gives the outline of a solution to these questions using a technique based on graph rewriting: a well-established technique for programming using graph-based abstractions [12]. We are working on developing tools for building generators for embedded systems [6], but it is our belief that the underlying technology general enough and it can be extended to provide an implementation of the ideas outlined in this paper.

Backgrounds

Design patterns are currently applied during development using a process that we can qualify as highly "manual." The typical process is as follows. First, the designer has to recognize the design problem that needs to be solved. This is often done such that the designer already knows a number of patterns, and type of problems they solve. He will match this knowledge with the particular application and its implied problems at hand. If there is a "good-enough" match between the two, then he applies the pattern in the context of the application. This often means tailoring or rewriting the example code known from the literature for the domain of the application, and perhaps adjusting already existing classes and their implementation with respect to the design pattern. To borrow a metaphor from aspect-oriented programming [7], this process is very much like "weaving in" the pattern into the current design. Obviously, there are activities here that can (and, probably should) not be made automatic. The designer is the best person to select a pattern and decide its applicability. However, it seems that the weaving process is a highly mechanistic manipulation of the source code, thus it can be the subject of automation. We believe that graph rewriting techniques can offer great support for not only implementing the pattern application, but also for the formal specification of patterns.

Graph rewriting is a technique of very high-level programming that grew out of the theory and application of graph grammars [13]. Graph grammars are extensions of formal (textual) languages into the realm of graphs. A generative (Chomsky) grammar gives a finite description of all the possible and syntactically correct sequences of symbols of a language in terms of terminal symbols, non-terminal symbols, production rules, and a start symbol. Note that the only composition allowed here is a linear concatenation: a sentence is always a linear sequence of symbols. In a graph language, the "sentences" are graphs composed of nodes and edges, and the composition operator is a graph connection: adding new nodes and edges to an existing graph. A graph grammar gives a generative description of all possible and syntactically correct graphs of a graph language.

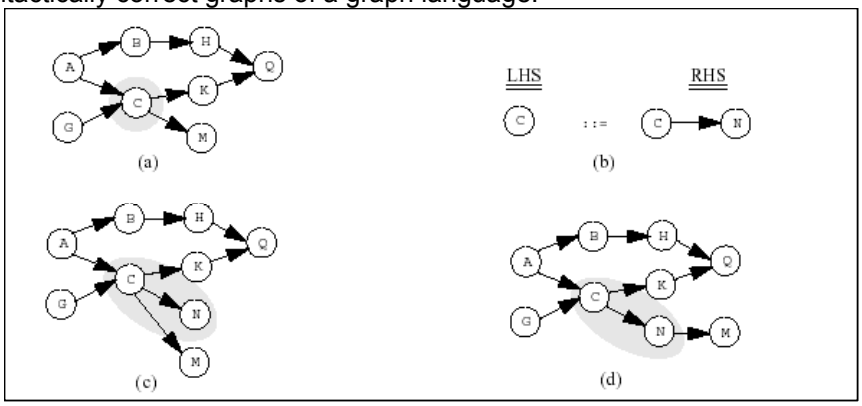

Figure 1: Example for graph rewriting

Graph rewriting borrows the production rule idea from graph grammars. A graph rewrite rule is a formal specification for substituting a subgraph with another graph. Figure 1 (borrowed from [12]) gives an example rewrite rule and its application. Section (a) shows a graph we want to rewrite using the rule specified in (b). Suppose the graph in (a) is a precedence network. Edges indicate task-completion constraints. For example, task C cannot begin until tasks A and G complete. The graph rewrite rule in (b) adds a new task N, to follow task C. This rule transforms the graph (a) into one of several possible results, such as (c) or (d). These results are different in the sense that they use different *embeddings* of the left hand side of the rule in the host graph (a). In general, the designer of the rewriting rule has to specify how the embedding should be done, and how the newly inserted subgraph should link up with the rest of the original.

Graph rewriting has been popular in the Computer Science community, especially in Europe, and a number of tools have been developed [13] to support programming via graph rewriting. The approach has not only a well-developed theory, but it also has been shown to be useful in a number of applications [14]. From the practical standpoint, graph rewriting is useful in the kinds of programming tasks where a function of a program can be expressed as the transformation of a graph into another graph. In these cases, one can express the transformation in the form of graph rewriting rules, and use a rewrite engine to execute the transformations. There have been tools developed that translate rules into executable code which performs the transformation in an efficient manner.
While graph rewriting is a very powerful technique, it obviously has some shortcomings, especially in terms of performance. It is inherently tied to a search process, which can be exponential in the worst case. Finding efficient techniques for improving the performance of graph rewriting tools is an active area of research.

Design patterns and graph rewriting

One can make a connection between design patterns and graph rewriting as follows. Let's assume that our design is expressed in the form of UML class diagrams [5], which capture domain specific classes, their attributes, and operations, and the various associations among them. When the designer introduces a new pattern into the design, he will add new or modify existing classes by adding new attributes, methods, associations, etc. The difference between the original design and the design with the pattern applied is, of course, a particular manifestation of the pattern. *Thus, we will consider pattern application as the application of a graph rewriting rule that converts the original design graph into a new design graph embellished with the pattern. We conjecture that design patterns can be represented as graph rewriting rules that operate on the class diagram of the design.*
There are a number of observations to be made here.

1. It is conceivable that not all patterns can be expressed as graph rewriting rules to be applied on class diagrams. We are not addressing those patterns here.
2. Pattern application is more than traditional graph rewriting; i.e., addition/deletion of nodes and edges in graph. Pattern application includes adding attributes, methods, weaving code into existing code, etc. Our hope is that, on a lower level, all these operations can be considered as actions performed during graph rewriting.
3. Pattern application is done on a special graph: the class graph of the application, not on some sort of instance graph. This is in concert with our assumption that the application (i.e., the design) is represented in a class diagram.
4. Pattern application is a design-time activity, and is performed when the engineer builds his design. The result of this graph rewriting is a new class diagram with the pattern's "code" weaved into the design. Obviously, graph rewriting can be applied in the application itself, during run-time, but that is a completely different issue.
5. The patterns as graph rewriting rules are written in terms of generic classes and generic associations, which are then matched against the specific classes of the class diagram. In this sense, patterns can be considered as generalized templates that span multiple classes, and whose code is generic, but will eventually be placed into a specific context.

A simple example for the Composite pattern is shown below in Figure 2. The composite pattern is taken directly from the GOF book [7], but it has been slightly extended. The pattern is encapsulated, and pattern input and output parameters are explicitly indicated.

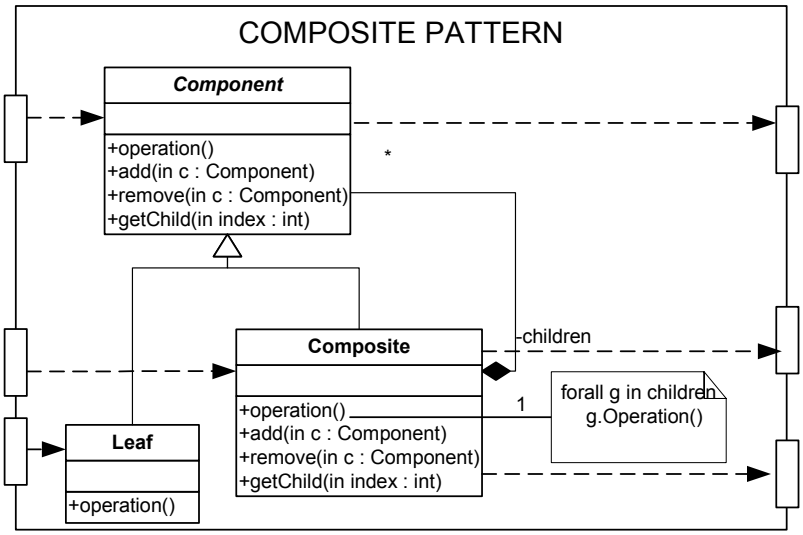

Figure 2: Composite pattern

The input and output parameters are like template parameters in C++: one can use the input parameters to specify what classes the pattern should be applied to, and use the output parameters to get the results of the pattern application process. Let's assume, in an application that we have a class `Primitive` with an operation called `run()`. Now if we want to apply the Composite pattern in this situation, and thus allow the formation of `Compound` classes (which contain `Compounds` or `Primitives` that support a `run()` operation) the application of the design pattern can be performed as shown in Figure **3** below.

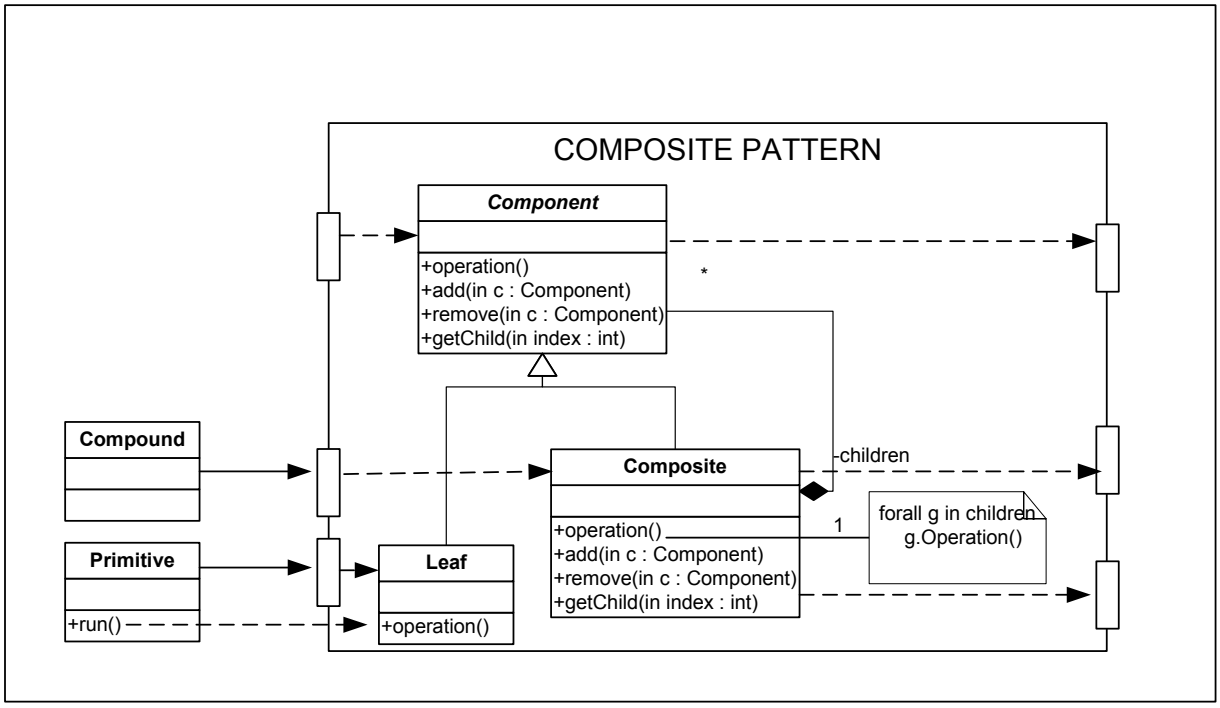

Figure 3: Applying the Composite pattern

The pattern application has the following semantics. The formal parameters of the pattern are bound to the actual parameters provided. On the diagram above, `Composite` of the pattern is bound to the `Compound` (an empty class), and `Leaf` is bound to `Primitive`. A parameter can be left unbound, like `Component` above. We also bind the `run()` operation of the `Primitive` to the operation of `Leaf`. The

application of the pattern works like the application of a graph rewriting rule: it will generate a new class diagram. The result of the application is shown below on Figure 4.

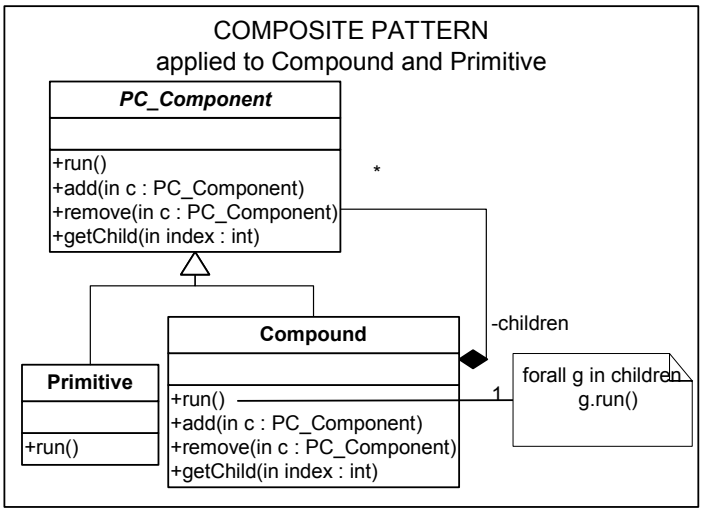

Figure 4: Composite pattern applied

The pattern application generates new classes, extends existing ones, inserts new associations, and new code. In general, the pattern application transforms an existing class graph into a new one, with new information weaved into the graph.

Extensions

The scheme introduced above obviously needs refinement, but it gives an initial approach for specifying and applying design patterns. One can envision an interactive tool, perhaps integrated into a UML modeling environment, like GME/UML [15] or Visio [16] that supports an incremental design process, where the designer applies patterns to an evolving design and creates new versions of the class diagram embellished with patterns. This process is recursive, with multiple patterns being applied to different classes, or applied on the results of previous pattern applications. Figure 5 below gives a notional picture of this pattern based development process.

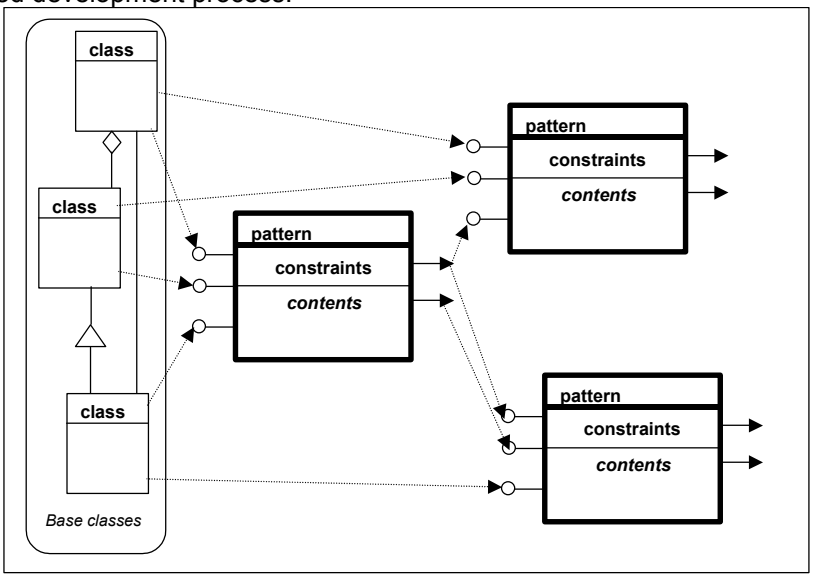

Figure 5: Applying multiple patterns

However, to turn this idea into a practical approach and to support a formal development process, a number of extensions need to be made to the basic scheme.
The first and foremost is the introduction of pre- and post-conditions in the patterns. The preconditions of a pattern describe what is expected from the classes (or class diagram fragment) the pattern is applied to. In general, it is some sort of constraint expression, which expresses this expectation using, for instance, a variant of first-order logic. A pattern application is valid only if the input arguments of the pattern satisfy the precondition. The post-conditions assert what conditions will be true for the result of the pattern application. As multiple pattern application can form a chain (as shown above), pre- and post-condition pairs can be evaluated and the global validity of the composition can be verified when the design is constructed. Perhaps the techniques of category theory (as introduced in [7]) can be used to verify the composition. Pattern composition might be reduced to a variant of type checking, but it seems that more powerful techniques might be necessary. Note that the conditions refer to the classes playing a role in the composition, so presumably the condition language can be made very simple.
In order to implement the approach described above in a tool, a development style has to be designed. We envision an interactive design environment, where the designer uses direct manipulation techniques to incrementally build a design. He introduces some initial classes, next applies patterns from a library to create new or extend the existing classes, etc. At any given time he has to interact with a design database, which keeps not only the final design but a living *design record*, which includes all pattern applications during the process, etc. The final result of the design activity is a class diagram that incorporates the design of the entire application. Composition checkers assist the designer by verifying pattern compositions using the "*assume-guarantee*" conditions associated with the patterns. Interestingly, this highly interactive style of development can be considered not only as a constructive activity, but also as an incremental analysis of the design. A mixed-mode, graphical/textual interface that allows direct manipulation is a necessity for an environment like this. In the background, the environment interacts with a graph rewriting engine, which applies the patterns and generates new class diagrams. During the construction process, explicit dependencies can be maintained, and changes propagated, as required.

Current and other work

We are working on a set of tools for building generators for embedded systems [6] that can efficiently transform components and their models, and models of component ensembles into code for running systems. These tools will allow the easy specification and customization of generators by sophisticated end-users, who want to create and possibly reuse their own generators, or any portion of those. In this project we are using graph rewriting technology as the implementation technique.
We believe that the same underlying graph rewriting technology can be used to support pattern application as discussed above. In fact, another project [7] uses these techniques to compose and implement middleware services for networked embedded systems.
There are a number of researchers who have introduced similar concepts recently. The techniques of generative programming [1] are important in the sense that they show how template-oriented composition can be used to implement complex software artifacts. The paper [18] describes a graph-rewriting based approach that uses meta-models, i.e., class diagram elements, to specify transformations on instance graphs. UMLAUT [19] is an extensible framework for UML diagrams. Their approach is based on a functional programming language, and (transformational) programs written in this language are used to represent patterns. The closest to our approach can be found in [20], but it is tied to the capabilities of the PROGRES graph programming tool.

Summary, conclusions, and future work

It is our belief that design patterns can, and must, have tool support in order to become even more widely used. To facilitate this, we offer the following —pragmatic— definition for design patterns:

> A design pattern is a parameterized collection of classes and associated code fragments with a well-defined behavior. A design pattern also specifies what preconditions it assumes and postconditions it asserts when applied in a particular context. Furthermore, non-functional consequences (the "costs") of applying a design pattern are also specified with the pattern.

Furthermore, we consider the parameterized collection of classes mentioned above as a *graph rewriting rule* that operates on a class diagram and applied at design time. We propose to introduce interactive

tools built on this foundation that engineers can use to incrementally construct a software design. We have shown on a simple example how a pattern application works, and how it could be used.
There are a number of research activities that have to be undertaken before this approach can be widely used. The precise semantics of the pattern application has to be specified, the pattern specification language has to be developed, the specific graph rewriting engine implemented, the assume-guarantee condition checking algorithms have to be developed, etc. just to name a few. However, once these activities are accomplished, pattern-based program development could be not only significantly enhanced, but it can also be made more formal.

Acknowledgement

The DARPA/ITO MOBIES program (F30602-00-1-0580) is supporting, in part, the activities described in this paper.

References

1. Czarnecki, K. Eisenecker, U: *Generative Programming - Methods, Tools, and Applications,* Addison-Wesley, 2000.
2. Richard P. Gabriel. Patterns of Software: tales from the software community. Oxford University Press. 1996.
3. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison-Wesley, 1995.
4. http://hillside.net/patterns
5. http://www.rational.com
6. http://www.isis.vanderbilt.edu/Projects/mobies/default.html
7. http://www.aosd.net/
8. http://www.serc.nl/people/florijn/work/patterns.html
9. http://www.research.ibm.com/journal/sj/budin/budinaut.html
10. http://iamwww.unibe.ch/~scg/Research/
11. http://www.kestrel.edu/HTML/prototypes/specware.html
12. Blostein, D., and Schurr, A. Computing with Graphs and Graph Rewriting. Software--Practice & Experience 29, 3 (1999), 1--21. 6
13. G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1,2,3. World Scientific, Singapore 1997-99.
14. D. Blostein, H. Fahmy, and A. Grbavec, "Practical Use of Graph Rewriting," Technical Report No. 95-373, Computing and Information Science, Queen's University, January,1995.
15. http://www.isis.vanderbilt.edu/Projects/gme/default.html
16. http://www.microsoft.com/office/visio/
17. http://www.isis.vanderbilt.edu/projects/nest/index.html
18. Lemesle R., Transformation rules bases on meta-modeling EDOC'98, San Diego, 1998
19. http://www.irisa.fr/UMLAUT/
20. Ansgar Radermacher: Support for Design Patterns Through Graph Transformation Tools. AGTIVE 1999: 111-126