Program Transformations for Evolving Software Architectures

Lance Tokuda
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188
unicron@cs.utexas.edu

1 Introduction

During the 1970s, evolution and maintenance accounted for 35 to 40 percent of the software budget for an information systems organization. This number jumped to 60 percent in the 1980s. Without a major change in approach, many companies may spend close to 80 percent of their software budget on maintenance by the mid-1990s [Pre92].

Object-oriented design methodologies offer important opportunities for reducing maintenance costs. For example, object-oriented software is organized into classes and frameworks that provide modularity and enhance reusability. This, in turn, can substantially reduce the cost of adding new functionality to a product. Language features such as inheritance also contribute to reuse and maintenance by allowing specializations of a class to be built without altering the original class.

Another kind of reuse is the reuse of designs. Commercial object-oriented languages such as C++ provide features to declare classes and relationships between classes (e.g., public and private inheritance) facilitating the construction and reuse of larger software artifacts. Recent work has recognized an important kind of design reuse: *object-oriented design patterns*, i.e., recurring patterns of relationships between classes, objects, methods, etc. that define preferred solutions to common object-oriented design problems [Coa92, Joh92, Gam94].

While design patterns are useful when included in an initial software design, they are often applied in the maintenance phase of the software lifecycle [Gam93]. For example, the original designer may have been unaware of a pattern or additional system enhancements may arise that require unanticipated flexibility. Alternatively, patterns may lead to extra levels of indirection and complexity inappropriate for the first software release.

We have discovered that some design patterns can be expressed as compositions of primitive program transformations. Moreover, we have noted that many of the transformations can be automated. Automating transformations would reduce the cost of certain kinds of program evolution, and eliminate programming errors and debugging that would otherwise have been introduced.

2 Design Patterns as Transformations

Design patterns capture expert solutions to many common object-oriented design problems: support for multiple implementations of a method, creation of compatible components, adapting a class to a different interface, subclassing versus subtyping, isolating third party interfaces, etc. Patterns have been discovered in a wide variety of applications and toolkits including Smalltalk Collections [Gol84], MacApp [App89], etc.

We have analyzed the design patterns in [Gam94] and we believe that program transformations can be applied to evolving systems to implement fifteen of the twentyfour patterns. This list is expected to grow as additional pattern transformations are identified. Furthermore, we have observed that these transformations are actually compositions of a small number of primitive object-oriented transformations.

Design patterns define target states for program transformations. Suppose an application **A** relied on the **MotifScrollBar** and **MotifWindow** classes. Now suppose **A** needs to be generalized to **Aprime** to use OpenLook widgets. The Abstract Factory design pattern [Gam94] defines the needed generalization. A useful transformation would convert the application code of **A** to that of **Aprime** by applying the Abstract Factory pattern to the **MotifScrollBar** and **MotifWindow** classes of **A**.

We believe that the implementation of design patterns in evolving systems can be accomplished using program transformations. In fact, we show in [Tok95] how an application comparable to **A** could be evolved to **Aprime** by applying a sequence of primitive program transformations.

3 An O-O Program Evolution Tool

Tools to execute these transformations would relieve users of tedious and error-prone tasks, and could drastically reduce application evolution and maintenance costs. Applying pattern transformations to evolving software systems can result in programs which are more general, extensible, and reusable than the original.

A key observation is that both the checks for satisfaction of pattern transformation preconditions and the actual code transformations performed in the example appear to be automatable. Automating transformations reduces the risk of introducing new errors when upgrading software. It also allows the user to perform selected changes quickly which can increase the speed at which evolution can take place.

A second observation is that it is often easier to view the effects of transformations at the class diagram level rather than by inspecting a list of source code differences. Class diagrams document classes and frameworks. Since most transformations evolve classes and frameworks, they are a useful method for documenting the effects of transformations. Class diagrams also provide a language independent means for representing changes that occur.

Choice of a language will affect the list of transformations provided. For example, some languages support factory methods directly. C++ was chosen as the first target language for this tool because of its widespread acceptance in industry. A primary focus of research will be to develop a complementary set of primitives which can be composed to form design patterns occuring in the target language.

4 Conclusions

The evolutionary phase of the software lifecycle can be expected to account for up to half of a software engineering organization's budget. Our approach to reducing cost is to automate as much of the evolutionary process as possible. This allows users to concentrate on essential design decisions and leaves the burden of low-level source code modifications to tools. Object-oriented transformations which evolve software architectures are a promising method that can be exploited.

We believe it is possible to build a tool which uses object-oriented transformations to implement design patterns in evolving systems. Required code changes are carried out automatically which reduces the risk of introducing new errors and facilitates rapid generalization and evolution of classes, frameworks, and programs. The result is an automated tool for producing more extensible and reusable software architectures.

References

[App89] Apple Computer Inc. *Macintosh Programmers Workshop Pascal 3.0 Reference.* Cupertino, California, 1992.

[Coa92] P. Coad. Object-Oriented Patterns. In *Communications of the ACM*, V35 N9, pages 152-159, September 1992.

[Gam93] E. Gamma et. al. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings, ECOOP '93*, pages 406-421, Springer-Verlag, 1993.

[Gam94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Gol84] Adele J. Goldberg, *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA, 1984.

[Joh92] R. Johnson. Documenting Frameworks with Patterns. In *OOPSLA '92 Proceedings*, SIGPLAN Notices, 27(10), pages 63-76, Vancouver BC, October 1992.

[Tok95] L. Tokuda and D. Batory, Automating Software Evolution via Design Pattern Transformations. In *3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico, October 1995.