IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Hershey PA 17033-1117, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4248

On Extracting the Semantics of the Iterator Pattern For Use in a Case Tool

[1]Joan Peckham and [2]Scott J. Lloyd
University of Rhode Island, [1]Tel: (401) 874-4174, [1]Fax: (401) 874-4617
[2]Tel: (401) 874-7056, [2]Fax: (401) 874-4312, joan@cs.uri.edu, sjlloyd@uri.edu

ABSTRACT

Software patterns are used to facilitate the reuse of object-oriented designs. While most CASE (Computer Aided Software Engineering) tools support the use of UML (Unified Modeling Language) [AO98] to extract the design from the software engineer and assist in development, most do not provide assistance in the integration and code generation of software patterns. In this paper, we analyze the Iterator software pattern [Gamma95] for the semantics that would be used in a CASE design tool to help the software engineer to integrate this pattern into the design and then generate some of the code needed to implement the pattern. This work is based upon semantic data modeling techniques that were previously proposed for the design of active databases [BMPV97, PMD95].

INTRODUCTION

One of the intents of the object-oriented (OO) programming paradigm is to assist in the reuse of code through the use of classes that bundle data structures and procedures in such a way that that they could more easily be moved from one implementation to another. When OO languages were first introduced, code libraries were developed to permit the sharing of objects and classes. At the same time, OOA&D (Object-Oriented Analysis and Design) techniques were being developed [Booch94, CY91, Jacobson92, Rumbaugh91, W-BWW90]. This gave us a set of notations for expressing the design of OO applications. The library became a vehicle for the reuse of the OO designs and led to the capture of software patterns or designs that are frequently reused in software applications, but are somewhat independent of particular application types. These patterns are archived in books, for example [Gamma95]. A combination of text, UML, and code samples is used to communicate the patterns. Early industrial experience indicates that patterns speed the development of systems but are hard to write [Beck96] so a few CASE tools provide computer assistance to the programmer in choosing and integrating automatically generated code for the patterns in their applications.

CASE tools that support the use of software patterns include [BFVY96, FMW97, Paulisch96]. While these tools are just beginning to emerge, none have integrated code generation and general design in a generic way that permits seamless code specification with patterns. For example, the techniques are not generally language independent and are unable to generate code in more than one language. Some existing tools generate code, but into a different workspace from the general software specification and coding environment, requiring the cutting and pasting of code from the pattern code space.

All software patterns have alternative implementations. These are typically explained using text and sample code. Software engineers are then expected to use this information to construct their own implementation of the pattern. Our goal here is to capture the semantics of patterns well enough so that they can be presented to the software engineer via a named choices in the CASE tool and then be used to generate the code. Earlier in [PM00] we began to elaborate the choices in the Observer pattern of [Gamma95]. In this paper we look at the Iterator pattern from the same source.[1]

A PATTERNS PRIMER

The Iterator Pattern

As laid out in [Gamma95], patterns are described using diagrams, textual descriptions, and sometimes with examples using a pseudo language or code. Included in each pattern description are:

Name - The name of the pattern

Classification - The type of pattern based upon the authors' classification system (there are three, creational, structural, and behavioral)

Intent - The purpose of the pattern, what it does

Also known as - Other used or known names for the pattern.

Motivation - A concrete example to help you understand the general pattern

Applicability - Where to use the pattern.

Structure - A graphical representation of the pattern using a UML like notation

Participants - Description of the participating objects and classes

Collaborations - How the participants interact

Consequences - How the pattern carries out its goals, and the results (positive and negative).

Implementation - Tips, possible pitfalls, and language specific issues.

Sample code - Some examples in a specific language.

Known uses - Where the pattern has been used in widely known commercial or research software.

These pattern characteristics help the reader to understand the nature of the pattern. From these patterns we can extract the alternative semantics for the pattern and use them to generate the code. In [Gamma 95] the Iterator pattern is described. The purpose of this pattern is to "provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." The intent here is to permit transparency in the access and use of the aggregate object. This permits the user of the aggregate to traverse it without having to know the details of its structure and implementation. This is a primary OO design principle that permits the encapsulation of particular modules of code to reduce errors and permit easy code maintenance. This decouples the aggregate object from the rest of the code and makes it easier to export it to other applications. At the same time, the aggregate is still useable by the application.

The basic structure of the Iterator pattern is shown in Figure 1. The client (the code that uses the Iterator) is shown in faint print. There are two roles in this pattern known as Aggregate and Iterator. The client must know of the existence of both. It calls the CreateIterator method in Aggregate to create Iterator, which then assists it to traverse Aggregate. But since this method is a part of the Aggregate class, the details of how to iterate and thus create the iterator are hidden from the client. The client however does also know that the methods First, Next, IsDone, and CurrentItem will be available once the Iterator is created.

Notice that the pattern makes use of abstract and concrete classes in the definition of the Iterator and Aggregate. This permits, for example, the creation of method signatures in the abstract class that are generic enough to be reused in a variety of applications and the concrete version with the elaborated code to be used in a specific application.

To implement the Iterator pattern the software engineer must normally read a textual description in a book, then develop a refined design, and then implement "by hand" the features of the pattern that are important for the intended application. For example:

Delete iterators- To prevent "memory leaks" caused by objects that are created and then not deleted once they are no longer needed in the program, there needs to be a means for

This paper appears in *Issues and Trends of Information Technology Management in Contemporary Organizations,* the proceedings of the Information Resources Management Association International Conference.

the client to delete the Iterator object. This is especially important in C++ where memory management is not supplied by the run-time program. Gamma et. al. [Gamma95] suggests that an IteratorPtr class is created to simplify the code needed to access the Aggregate object and to assure that a destructor is called to remove the object when it is no longer used. Most of the code can be generated when the Iterator pattern is chosen. Thus, the software engineer saves time and prevents errors by not having to code this feature each time the pattern is used.

The above is a feature that should always be used when coding the pattern in C++. In the next example, we look at pattern options that would need alternative implementations for the same feature (iteration). In some cases, different code can be generated, therefore, the CASE tool provides the software engineer with a means to choose the specifics of a particular implementation:

List Iteration Choices

There are two choices for iterating over the aggregate object, *internal* (II) and *external (EI)*. If the client controls the iteration by explicitly requesting the next element from the iterator, then this is external. The other alternative is that the iterator carries out the whole iteration once the client begins the process. The intended application will of course determine this choice and the software engineer should merely implement the appropriate method. Once this is chosen, there are additional alternatives for implementation. For example, if generating code in a language that does not have adequate support for the parameterization of the functions used to iterate, the internal iteration alternative can be clumsy to implement; however, there are two choices available to the engineer: 1) Pass a pointer to the function needed to be applied iteratively, and 2) Use inheritance to resolve the particular functions needed.

The code for applying the internal iterator is given in Gamma, although knowing the type of aggregate object upon which the iteration is being carried out as well as the type of activity performed during iteration is needed. In the example given, the aggregate object was a list of employees and the operator applied upon iteration was PrintEmployees. However, the code generation facility in the CASE tool can generate the code and leave the specific aggregate and operators blank with instructions for the software engineer to fill in. For example, here is a passage of C++ code needed to implement an internal iterator:

```
template <class Item>
class ListTraverser {
public:
        ListTraverser (List <Item>* aList);
Bool Traverse ();
protected:
        Virtual bool ProcessItem (const Item&) = 0;
private: ListIterator<Item> _iterator;
};
```

This code can be generalized and given slots to be filled in for the particular application in question, for example the software engineer can be presented with the following with a legend that annotates the code with a guide of how to fill in the italicized code:

```
template <class AggItem>
class AggTraverser {
public:
        AggTraverser (Agg <AggItem>* AggType);
Bool Traverse ();
protected:
        Virtual bool ProcessAggItem (const AggItem&) =
        0;
private: AggIterator<AggItem> _iterator;
};
```

In some cases, the implementation choices are not very specific and very little code can be generated. But, even if only class and method headers are generated, this can assist the programmer in making sure that all of these features are present.

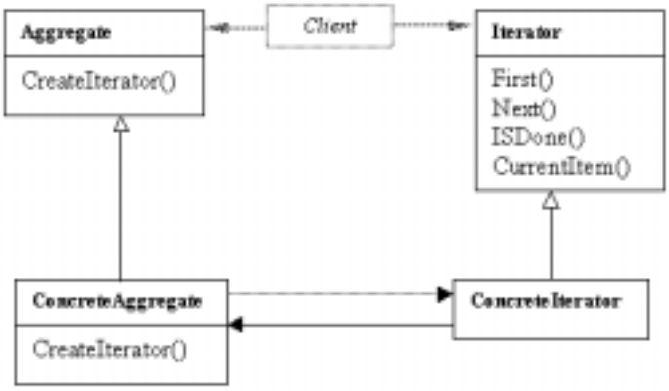
CONCLUSIONS

Software patterns can be very useful to prevent software designers and developers from "reinventing the wheel" for every new utilization of a specific pattern . This can lower the cost of software development and maintenance, providing more robust and error free code. Currently software patterns have not been integrated into CASE tools in a robust manner. In this paper we have shown how we envision the integration of patterns into CASE design tools assist in the selection of the proper patterns and to partially generate the code to implement the Iterator pattern. There are numerous other applications possible for this type of software tool that will be explored in future research. For example, the pattern books outline how specific patterns can be used with each other in a system design. The CASE tool can use this information to offer assisting and cooperating patterns to the designer and begin to generate the code. It can also provide checking assistance in identifying problems or errors that are frequently made when specific patterns or combinations of patterns are selected.

ENDNOTE

1 Thank you to Heng Chen, Chen Gu, and Mingsong Zheng of Professor Peckham's CSC 509 (Software Engineering) class for beginning to outline the semantics of this pattern in a classroom exercise for us.

Figure 1: Iterator pattern, basic structure



REFERENCES

[AO98] Alhir, A., Oram, A. *UML in a Nutshell: A Desktop Quick Reference (Nutshell Handbook)*. O'Reilly and Associates. 1998.

[Beck96] Beck, D., et. al., "Industrial Experience with Design Patterns", *Proceedings of ICSE-18,* IEEE, 1996, p. 103-113.

[Booch94] Booch, G., *Object Oriented Analysis and Design,* 2nd ed., Benjamin Cummings, 1994.

[BMPV97] Brawner, MacKellar, Peckham, and Vorbach, "Automatic Generation of Update Rules to Enforce Consistency Constraints in Design Databases'', *in 7th IFIP 2.6 Working Conference on Database Semantics (DS-7) Searching for Semantics: data mining, reverse engineering, etc*., Spaccapietra and Maryanski, editors, Chapman and Hall, 1997.

[BFVY96] Budinsky, F., Finnie, M., Vlissides, J., and Yu, P., "Automatic Code Generation from Design Patterns", IBM Systems Journal, Vol.35, No. 2, p. 151-171, 1996.

[CY91] Coad, P, and Yourdon, E, *Object-Oriented Analysis,* 2nd ed., Prentice-Hall, 1991.

[FMW97] Florijn, G, Meijers, M., van Winsen, P., Tool Support for Object-Oriented Patterns, Proceedings of ECOOP'97, Finland, 1997.

[Gamma95] Gamma, Helm, Johnson, Vlissides, and Booch. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.

[Jacobson92] Jacobson, I. *Object-Oriented Software Engineering,* Addison-Wesley, 1992.

[Paulisch96] Paulisch, F., "Tool Support for Software Architecture", SIGSOFT 96 Workshop, San Francisco, CA, 1996, p. 98-100.

[PMD95] Peckham, MacKellar, Doherty. "A Data Model for the Extensible Support of Explicit Relationships in Design Databases", *VLDB Journal,* 4(2), 1995, p. 157-159.

[PM00] Peckham and MacKellar, "Generating Code for Engineering Design Systems Using Software Patterns", *Artificial Intelligence in Engineering,* Elsevier, 2000.

[Rumbaugh91] Rumbaugh, J., et.al., *Object Oriented Analysis and Design,* Prentice-Hall, 1991.

[W-BWW90] Wirfs-Brock, Wilkerson, and Weiner, L., *Designing Object-Oriented Software,* Prentice-Hall, 1990.