Framework Development and Reuse Support

Wolfgang Pree
C. Doppler Laboratory for Software Engineering
Johannes Kepler University of Linz, A-4040 Linz, Austria
Voice: ++43 70-2468-9432; Fax: ++43 70-2468-9430
E-mail: pree@swe.uni-linz.ac.at

Abstract

Although object-oriented programming techniques have evolved into an accepted technology with recognized benefits for software development, conventional text-based programming languages together with state-of-the-art browsing tools seem to be insufficient for a software engineer to cope with the complexity of class libraries, especially application frameworks. Both visualization techniques and visual programming seem appropriate to partially overcome these shortcomings of object-oriented programming.

Software development based on an application framework requires learning its high-level *language*. This contribution discusses concepts and corresponding tools to support the learning of such a language: a design book visualizes the design of an application framework; an active cookbook guides the programmer through typical adaptation steps.

Keywords:
design visualization, object-oriented programming, class libraries, application frameworks, visual manipulation

1 Introduction

Concepts offered by object-oriented programming languages are often used to produce reusable single components. The concepts of inheritance and dynamic binding are sufficient to construct application frameworks, i.e., reusable semifinished architectures for various application domains. Such frameworks represent a real breakthrough in software reusability: not only single building blocks, but also the design of (sub)systems can be reused.

Application frameworks such as Smalltalk's MVC framework [Krasner 1988], MacApp [Schmucker 1986; Wilson 1990], AppKit [NeXT 1990] and ET++ [Weinand 1988; Gamma 1992; Eggenschwiler 1992] can be viewed as the first test bed for the development of reusable architectures by means of object-oriented concepts. These frameworks for the graphic user interface (GUI) domain have become one of the main reasons that object-oriented programming enjoys such a good reputation for promoting extensibility and reuse. Frameworks are not restricted to a narrow range of domains, but are almost universally applicable. Examples are frameworks for VLSI routing algorithms [Gossain 1989] and operating systems [Russo 1989].

Main thesis. Frameworks amount to a real breakthrough in software reusability if they can be adapted easily to the specific needs of a domain. Interactive visual techniques

discussed in this contribution help in understanding, refining and developing frameworks.

A *framework* defines a high level language with which applications within a domain are created through specialization. *Specialization* takes place at points of predefined refinement that we call *hot spots*. A framework-centered software development process requires the creation and reuse of frameworks. We recommend new tools—electronic books—to support this process:

- *Creation of frameworks*: *Design books* communicate the design of existing frameworks so that some design ideas can be reused in the creation of new frameworks.
- *(Re)use of frameworks*: Design books and *active cookbooks* assist in specializing a particular framework.

Appropriate design books and active cookbooks should help an experienced programmer understand internals of frameworks, and develop software based on these reusable architectures. Serving as an aid for programming novices to support end user computing is a secondary goal of these electronic books.

Since we can develop application frameworks for a range of domains, the concepts and tools we propose are applicable to various levels of programming. For instance, design books and active cookbooks can be provided for application frameworks dealing with low-level operating system tasks as well as for application frameworks for high-level software, such as reservation systems.

2 Conceptual and terminological foundations

Terms are often misused in the realm of object-oriented software development. This section builds a terminological and conceptual basis for the paper.

Application framework. An *application framework* consists of ready-to-use and semifinished building blocks. The overall architecture, i.e., the composition and interaction of building blocks, is predefined as well. Producing specific applications usually means adapting components to specific needs by implementing some methods in subclasses of application framework classes.

In general, an application framework standardizes building blocks for a specific domain. An application framework is well-designed if it offers flexibility where required, i.e., it can easily be adapted to domain-specific requirements. For example, ET++ is a well-designed application framework for the GUI domain. [Weinand 1989] states that writing an application with a complex GUI by adapting ET++ can result in a significant reduction in source code size (i.e., the source code that has to be written by the programmer who adapts the framework) compared to software written with the support of a conventional graphic toolbox.

Usually the term application framework expresses that its building blocks constitute a generic application for a domain. It is often hard to decide whether a framework migrates to this category. We use the terms framework and application framework synonymously. Only when a distinction is necessary, we use the term application framework explicitly.

Abstract classes and abstract methods. The principal idea underlying frameworks can be subsumed in the following way. Semifinished components of a framework are represented by *abstract classes*. Some methods of an abstract class can be implemented, while only dummy or preliminary implementations can be provided for other methods, called *abstract methods*.

As a consequence, abstract classes are not instantiable classes. Their purpose is to *standardize the class interface* for all subclasses—subclasses can only augment the

interface, but not change the names and parameters of methods defined in a superclass. Instances of subclasses of an abstract class will understand at least all messages that are defined in the abstract class. The term *contract* is used for this standardization property: instances of subclasses of a class *A* support the same contract as supported by instances of *A*.

The implication of abstract classes is that other semifinished or ready-to-use components of a framework can be implemented based on the contract of an abstract class. In the implementation of these components, reference variables are used that have the static type of the abstract classes they rely on. Polymorphism allows such components to work with instances of subclasses of an abstract class. Due to dynamic binding, the concrete instances of subclasses of an abstract class bring in their specific behavior. So the behavior of framework components is ideally adapted by implementing only the abstract methods of semifinished components. The abstract methods constitute the principal *hot spots* of a framework.

Design patterns and metapatterns. It is still a matter of dispute how to describe essential design aspects in object-oriented systems. *Design pattern* approaches recently emerged in the object-oriented community to cope with this problem. Proposed design pattern catalogs such that of Erich Gamma, et al. [Gamma 1992; Gamma 1994] try to describe frameworks on an abstraction level higher than the corresponding code that implements these frameworks.

Design pattern catalogs essentially attempt to pick out frameworks that are not too domain-specific. Such frameworks are presented as examples of good object-oriented design that can be applied in the development of other frameworks. For example, the model/view aspect of the MVC framework constitutes a pattern in the design pattern catalog. In this miniframework, a model component notifies its dependent view components when changes occur. The view components display the data represented by the model component. Thus, view components have to be informed to update in case of model changes.

Design pattern catalogs containing specific framework examples are useful means to construct new frameworks. Nevertheless, a more advanced abstraction is necessary. This is useful to actively support the design pattern idea and to visualize a framework's design.

We use the term *metapatterns* for a set of design patterns that describes how to construct frameworks independent of a specific domain. These metapatterns prove to be a simple yet powerful approach that can be applied to categorize and describe any specific framework pattern on a metalevel, so that the hot spots can be identified immediately.

Metapatterns presented in Section 3 do not replace state-of-the-art design pattern approaches, but rather metapatterns complement these approaches. Design books discussed in detail in Section 4 are based on these metapatterns.

Design books and active cookbooks. *Design books* help the user browse through the essential design of a framework to see how semifinished components gain their flexibility. They are a way to visualize these flexible hot spots.

Design books can be viewed as advanced design pattern catalogs. Some aspects of a specific framework might be domain-independent, so that this design can be applied in the development of new frameworks. In these cases, design books serve the same purpose as design pattern catalogs. Actually, design pattern catalogs can be viewed as carefully chosen subsets of the design examples that can be captured and categorized in design books.

In addition to design pattern catalogs, design books can document the design of any domain-specific framework. Because metapattern browsers allow efficient design

documentation of frameworks, they can help in adapting the hot spots of a framework to specific needs.

Application framework *cookbooks* contain numerous *recipes*. These recipes describe in an informal way how to adapt a framework to specific needs within a domain. Recipes usually do not explain internal design and implementation details of a framework.

Recipes are rather informal documents. Nevertheless, most cookbook recipes are structured roughly into the following sections:

- purpose
- steps: how to do something, including references to other recipes
- source code example(s)

A programmer has to find the recipe that is appropriate for a particular framework adaptation. This recipe is then used by simply adhering to the steps that describe how to accomplish a certain task.

We envision more active support of this typical way of developing software based on a framework, and call the corresponding tool an *active cookbook*.

Active cookbooks and design books support the reuse and development of frameworks. We first present metapatterns that form the conceptual basis of design books and go on to discuss these electronic books.

3 Metapatterns

Metapatterns point out the few essential ideas of how to construct frameworks, i.e., reusable, flexible object-oriented software architectures, independent of a specific domain. They constitute an elegant and powerful approach to describe and visualize the design of a framework. Such an efficient design communication helps develop and reuse frameworks.

3.1 Class/object interface and interaction metapatterns

Template and hook methods represent the metapatterns required to design frameworks consisting of single classes and/or groups of classes together with their interactions. A template method implements a functionality in a generic way. Hook methods called from a template method form its generic parts. Subclasses of the class where a template method is defined can change the behavior of this template method at these predefined spots by implementing hook methods in a specific way.

The terms “template method” and “hook method” are commonly used by various authors, e.g., [Wirfs-Brock 1990; Gamma 1994]. We assume that all methods are dynamically bound. (For example, this assumption is not valid for C++.) We use the notation proposed by [Rumbaugh 1991] in order to depict class diagrams.

Let us take a closer look at how framework concepts are applied at the microlevel, i.e., in the implementation of methods. In order to implement frameworks, we have to consider the following kinds of methods:

- *template methods* that are based on
- *hook methods*, which are either
  - *abstract methods*, or
  - template methods

Let us consider the example shown in Figure 1: We assume that an abstract class *RentalItem*, which could be part of an application framework for reservation systems,

offers the three methods *PrintInvoice()*, *CalcRate()* and *GetName()*. *PrintInvoice()* constitutes the template method based on the hook methods *CalcRate()* and *GetName()*. The hook methods are abstract methods. Abstract classes and abstract methods are written in *italic style* in the graphic representation.

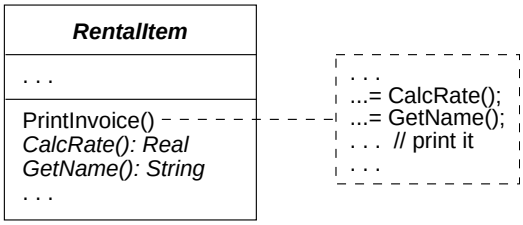

Figure 1 Template method *PrintInvoice* based on hook methods *CalcRate* and *GetName*

The adaptation of *RentalItem* has to be done in a subclass where the abstract methods *CalcRate()* and *GetName()* of *RentalItem* are implemented. The default implementation of *PrintInvoice()* meets the requirements of the specific application under development.

The template method *PrintInvoice()* is adapted, for example, in a subclass of *RentalItem* called *HotelRoom* without changing the source code of *PrintInvoice()* (see Figure 2a). Figure 2b illustrates schematically the hook methods as hot spots of the template method *PrintInvoice()*. Framework specialization takes place at these hot spots.

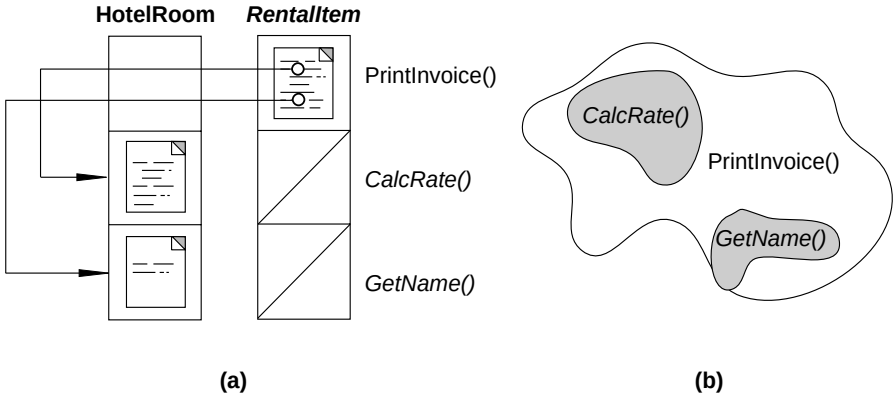

Figure 2 Adaptation of template method *PrintInvoice()*

3.2 Class/object composition metapatterns

In the example shown in Figure 1 and Figure 2, hot spots (the hook methods) and frozen spots (the template methods) are unified in one class. This implies that the behavior of printing an invoice can be changed by defining a subclass and implementing its hot spots in a specific way.

In some situations, more flexibility is required to allow adaptations at run time. In order to achieve this degree of flexibility, frozen spots and hot spots have to be put into separate classes as explained below. In this case, the class that contains the hook method(s) can be considered as the *hook class* of the class that contains the corresponding template method(s). We call the class that contains the template method(s)

template class. In other words, a hook class *parameterizes* the corresponding template class; the hot spots are in the hook class. We use the letter *T* for template class and *H* for hook class. We refer to instances of *T* and *H* or their subclasses as *T* objects and *H* objects.

Adaptations at run time become possible since the behavior of a *T* object, i.e., its template method, can be changed by associating a different *H* object with the *T* object. Creating *H* objects and assigning references to *T* objects can, of course, be done at run time.

The following example illustrates the case where class *RentalItem* is based on the contract of class *Calculator* (see Figure 3). So how a rental item prints an invoice can be changed at run time by assigning a different object to the instance variable *calculator* of a *RentalItem* object, one that fulfills the contract of *Calculator*. How *RentalItem* objects and *Calculator* objects are coupled via *calculator* is called *abstract coupling*.

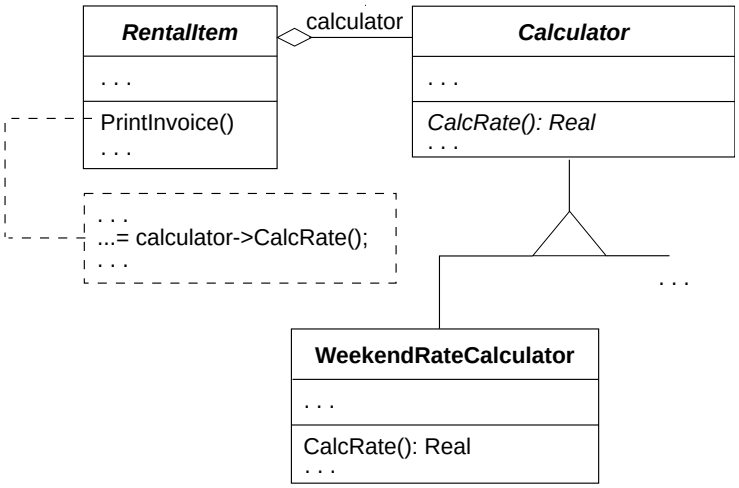

Figure 3 Abstract coupling based on template and hook methods

What constitutes a template method and a hook method, and so a template class and a hook class, depends on one's point of view. A hook method is elementary compared to the template method in which the particular hook method is used. In another context, the template method can become a hook method of another template method.

For example, an instance of a class *RentalItemManager* manages an arbitrary number of *RentalItem* objects. A template method *PrintInvoices()* sends the message *PrintInvoice* to some of the managed *RentalItem* objects, depending on a certain condition. So in this context *PrintInvoice()* of class *RentalItem* becomes the hook method of the template method *PrintInvoices()* in class *RentalItemManager*. The behavior of *RentalItemManager* might be required in a hotel reservation system, for example, to print invoices for a tourist party leaving the hotel. Figure 4 shows the relevant aspects of the classes *RentalItemManager* and *RentalItem*.

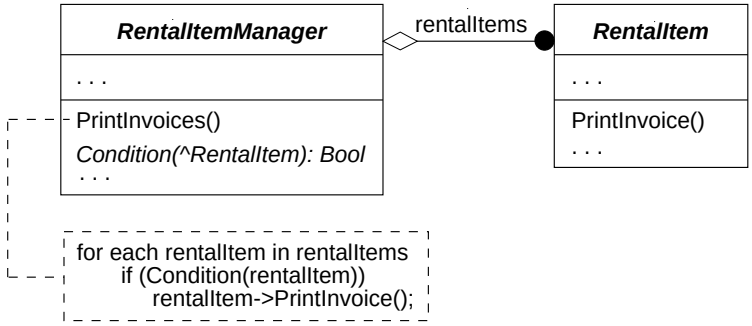

Figure 4 *RentalItem* as hook class in connection with template class *RentalItemManager*

In general, it is interesting to consider how *T* and *H* objects can be composed. We term the few combination possibilities *composition metapatterns*.

The simplest composition metapattern is the unification of *T* and *H* in one class. This special case is termed *Unification metapattern*, which underlies class *RentalItem* (see Figure 1), for example.

If *T* and *H* are separated, a *T* object might refer to exactly one *H* object or an arbitrary number of *H* objects. Figure 5 depicts the corresponding *1:1 Connection metapattern* and *1:N Connection metapattern*. The 1:1 Connection metapattern underlies the miniframework in Figure 3; the 1:N Connection metapattern the one in Figure 4.

Figure 5 1:1 Connection metapattern (a) and 1:N Connection metapattern (b)

If *T* is a subclass of *H*, recursive object composition becomes possible as demonstrated in Figure 6 for the *1:N Recursive Connection metapattern*. Note that the names of template and hook methods are typically the same in the 1:N Recursive Connection metapattern, i.e., *TH()*.

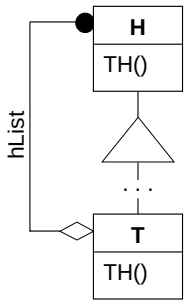

Figure 6 1:N Recursive Connection metapattern

Since *hList* manages objects of static type *H*, objects of any subclass of *H*, i.e., also *T* objects can be handled. This allows building up directed graphs with *T* objects and *H* objects as nodes. Figure 7 shows a tree as an example.

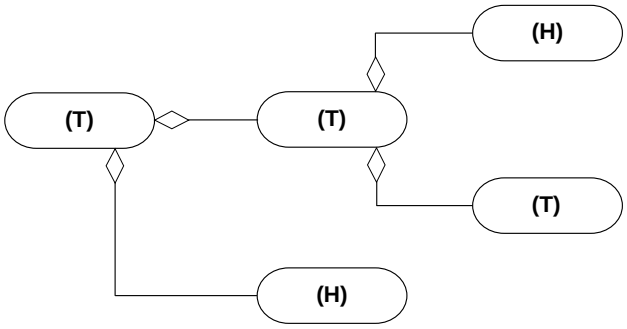

Figure 7 Tree hierarchy of *T* and *H* objects

The typical structure of *TH()* in class *T* is analogous to the template method *PrintInvoices()* shown in Figure 4. The message *TH* is forwarded to the managed *H* objects, probably depending on a condition. So the following characteristic of object composition based on the 1:N Recursive Connection metapattern results: A *T* object can be viewed as a place holder for all objects following that *T* object in the directed graph. Instead of sending the message *TH* to all these objects, it is sufficient to send the message to the particular *T* object. This message is then automatically forwarded to the other objects that are placed "behind" that *T* object in the directed graph.

Due to this forwarding property of typical template methods in recursive connection patterns, *a hierachy of objects* built by means of the 1:N Recursive Connection pattern *can be treated as a single object.*

When to choose other recursive composition metapatterns is discussed in detail in [Pree 1994].

To sum up, hook methods/classes represent the requirements necessary to produce a specific application for a domain out of a semifinished application framework. Framework-centered software development requires us to *see* these hot spots. The subsequent section discusses how hot spots can be visualized, based on metapatterns.

4 Metapatterns as basis of design books

Each composition metapattern typically occurs several times in a framework. Only the semantic aspect of the hot spots differs. The characteristic of a metapattern, especially the offered degree of flexibility, is independent of the particular situation where a metapattern is applied.

The principal idea of a design book is to attach composition metapatterns to framework components. A design book for a framework constitutes a means to browse through its hot spots. Depending on the characteristics of a metapattern, the programmer *sees* the intended adaptations and their corresponding degree of flexibility.

For example, the Unification metapattern could be attached to various components of a framework. Figure 8 illustrates this for class *RentalItem*.

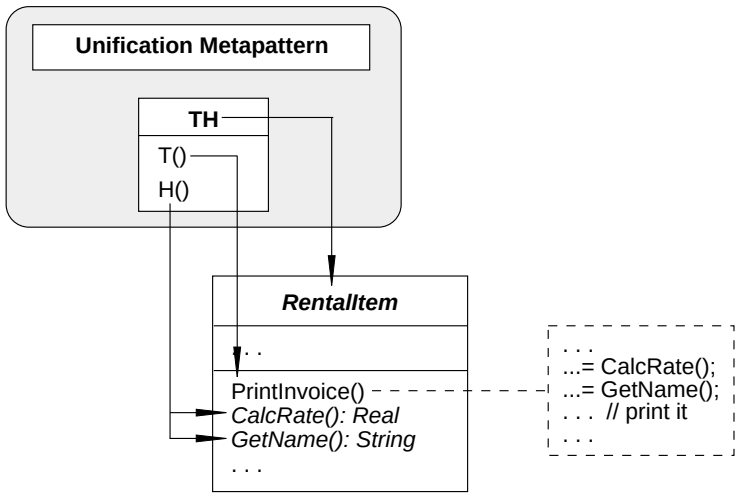

Figure 8 Attaching the Unification metapattern to class *RentalItem*

Analogously, the other composition metapatterns can be attached to framework components. Figure 9 demonstrates this for the 1:1 Connection metapattern linked to *RentalItem* and *Calculator*.

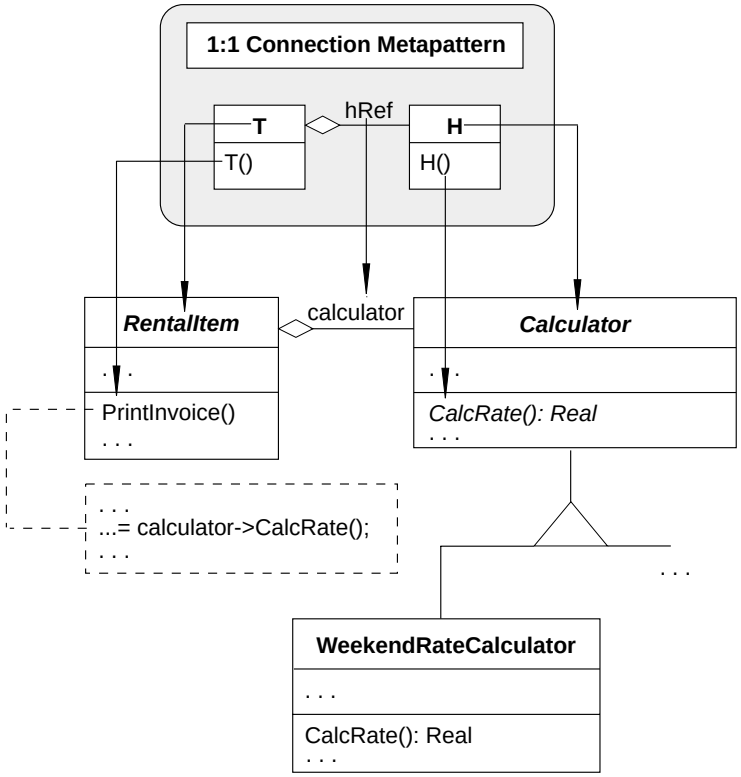

Figure 9 Attaching the 1:1 Connection metapattern to class *RentalItem* and class *Calculator*

A hypertext system is well suited for implementing a design book. The arrows in Figure 8 and Figure 9 would be hypertext links in a design book. Figure 8 and Figure 9 assume that there exists a class diagram for the annotated components. Thus the links refer to the corresponding items in that class diagram. If no such class diagram is available, links could refer directly to the corresponding source code fragments.

Since a particular metapattern occurs several times in a framework, a design book has to handle this in an appropriate way. Due to the fact that only the semantic aspect of a metapattern differs, we propose a solution as sketched in Figure 10.

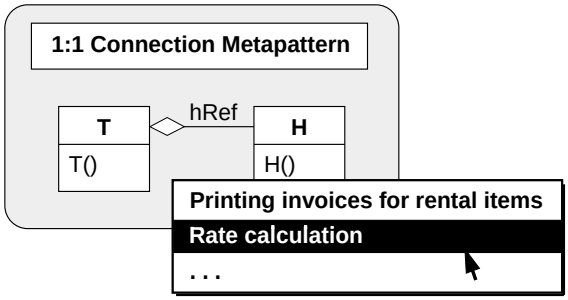

Figure 10 Linking specific examples to metapatterns

A menu pops up if a design book user clicks inside the hook class of a metapattern. The hook class represents the hot spot in the pattern. The menu lists the various aspects that

are kept flexible in a framework and that are based on the 1:1 Connection metapattern. If the item “Rate calculation” is chosen, the corresponding example is linked to the metapattern. In this case, it would be the components as depicted in Figure 9.

An appropriate hypertext editor that lets us define links between metapatterns and source code and/or class diagrams, forms the basis for production of design books for any application framework. Since there will be myriads of metapatterns in (application) frameworks, programmers who know a particular framework well, especially its developers, should produce the corresponding design book.

The fact that the hot spots are visualized in a design book might be useful not only for adaptations of a particular framework. Some aspects of a framework might be pretty domain-independent, so that a programmer who grasps the design can apply it in the development of other frameworks.

5 Active cookbook

Conceptually, an active cookbook differs from a design book in that it focuses on implementation details of an adaptation. Design books outline the hot spots for specialization on an abstraction level that is higher than the underlying implementation details.

Once the hot spots for an adequate framework specialization are identified by means of a design book, an active cookbook helps to accomplish the adaptation steps. Though design books and active cookbooks are conceptually different, they are predestined to be integrated in one tool.

The concept of actively assisting in the adaptation of frameworks is already applied in tools in the realm of GUI application frameworks. For example, ParcPlace’s VisualWorks, NeXT’s InterfaceBuilder, and the “Wizard” tools of Microsoft’s Foundation Classes actively support the adaptation of the corresponding GUI application framework.

The idea of actively supporting framework adaptations should be applied to application frameworks for any domain. For example, to define a subclass of *Calculator*, an active recipe would provide an editor that asks the programmer for the name of the subclass. After providing the calculation algorithm—either textually or, if appropriate, through visual manipulations—the corresponding class is generated.

Simple configuration tasks such as choosing which *Calculator* object is combined with a *RentalItem* object could be specified by selecting from a list as shown in Figure 11. Note that such configuration tools can also be handled by end users. Metapatterns that allow run time adaptations are predestined for such tools, assuming that some ready-to-use *H* components exist already. In the example shown in Figure 11, specific subclasses of the hook class *Calculator* have already been defined and implemented.

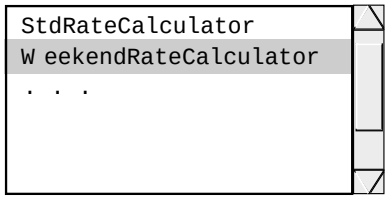

Figure 11 Part of a configuration dialog

6 Summary

We are convinced that visually supported programming based on application frameworks should not primarily rebuild concepts already provided by textual object-oriented programming languages. A general-purpose visualization tool like a design book, together with an active cookbook that integrates specific visual manipulation editors, supports the reuse and development of application frameworks by visualizing hot spots. The textual object-oriented language is used for object definition, message passing and control flow specification.

In order to evaluate the design book and active cookbook approach, prototypes of these tools were implemented[1]. The prototypes are based on a framework that simulates an ubiquitous computing world [Weiser 1991].

Future research and hands-on experience with the design book and active cookbook prototypes will reveal pros and cons of the presented vision. Research is especially necessary to define generic editors that can be integrated into active cookbooks for several different domains.

Research and experience will also show whether a (more or less) pure visual specification of dynamic aspects of an object-oriented software system is superior to textual languages. In this case, appropriate visualization concepts could be integrated into the electronic books.

Acknowledgements

Erich Gamma, one of the developers of ET++, did pioneering work in his PhD thesis [Gamma 1992], which uses a graphic notation together with an informal textual representation as a basis for describing the design of ET++. This way of describing object-oriented design on an abstraction level higher than the underlying object-oriented programming language stimulated the search for metapatterns.

I thank Prof. Takayuki Dan Kimura from Washington University in St. Louis. During my stay at WashU, numerous discussions on visual programming provided essential insights regarding visual programming and especially the motivation to find ways to combine object-oriented and visual programming paradigms.

Prof. Gustav Pomberger and the team at Siemens AG Munich provided the necessary environment to discuss the presented ideas and to develop the corresponding prototypes. Albert Schappert provided helpful comments on earlier versions of this contribution.

Adele Goldberg’s detailed hints helped to bring out the core points in a final revision.

---

[1] This research is carried out in cooperation with Siemens Corporate Research in Munich and the PenLab at Washington University in St. Louis.

References

[Eggenschwiler 1991] Eggenschwiler T.: Design Support in a Very Large Class Library: A Bottom-Up Approach; Semesterarbeit, Institut für Informatik, University of Zürich, 1991 (in order to obtain a copy write to: Institut für Informatik, Universität Zürich, Winterthurerstr. 191, CH-8057 Zürich, Switzerland).

[Eggenschwiler 1992] Eggenschwiler T., Gamma E.: ET++ Swaps Manager: Using Object Technology in the Financial Engineering Domain; OOPSLA'92, Special Issue of SIGPLAN Notices, Vol. 27, No. 10, 1992.

[Gamma 1992] Gamma E.: Objektorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design; doctoral thesis, University of Zürich, 1991; published by Springer Verlag, 1992.

[Gamma 1994] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns—Microarchitectures for Reusable Object-Oriented Software (preliminary title); to be published by Addison-Wesley in 1994.

[Gossain 1989] Gossain S., Anderson D.B.: Designing a Class Hierachy for Domain Representation and Reusability; Proceedings of Tools '89, Paris, France, 1989.

[Krasner 1988] Krasner G.E., Pope S.T.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80; Journal of Object-Oriented Programming, 1, 3, 1988.

[NeXT 1990] NeXT, Inc.: 1.0 Technical Documentation: Concepts; NeXT, Inc., Redwood City, CA, 1990.

[Pree 1994] Pree W.: Design Patterns for Object-Oriented Software Development (preliminary title); to be published by Addison-Wesley/ACM Press in 1994.

[Rumbaugh 1991] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: Object-Oriented Modeling and Design; Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Russo 1989] Russo V., Campbell R.H.: Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierachical Design; in Proceedings of OOPSLA '89, New Orleans, Louisiana, 1989.

[Schmucker 1986] Schmucker K.: Object-Oriented Programming for the Macintosh; Hayden, Hasbrouck Heights, New Jersey, 1986.

[Weinand 1988] Weinand A., Gamma E., Marty R.: ET++ - An Object-Oriented Application Framework in C++; OOPSLA'88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, 1988.

[Weinand 1989] Weinand A., Gamma E., Marty R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework; in Structured Programming Vol.10, No.2, Springer 1989.

[Weiser 1991] Weiser M.: The Computer for the 21st Century; Scientific American, September 1991.

[Wilson 1990] Wilson D.A., Rosenstein L.S., Shafer D.: Programming with MacApp; Addison-Wesley, 1990.

[Wirfs-Brock 1990] Wirfs-Brock R., Wilkerson B., Wiener L.: Designing Object-Oriented Software; Prentice Hall, Englewood Cliffs, New Jersey, 1990.