Meta Patterns—A Means For Capturing the Essentials of Reusable Object-Oriented Design

Wolfgang Pree

C. Doppler Laboratory for Software Engineering
Johannes Kepler University Linz, A-4040 Linz, Austria
Voice: ++43 70-2468-9431; Fax: ++43 70-2468-9430
E-mail: pree@swe.uni-linz.ac.at

Abstract. There is an undeniable demand to capture already proven and matured object-oriented design so that building reusable object-oriented software does not always have to start from scratch. The term *design pattern* emerged as buzzword that is associated as a means to meet that goal. Already existing approaches such as the catalog of design patterns of Erich Gamma et al. [5, 6] and Peter Coad's object-oriented patterns [3] differ in the applied notation as well as the way of abstracting from specific application domains.

This paper proposes a domain-independent terminology and notation we call meta patterns. It is demonstrated how meta patterns constitute a minimal means to capture reusable object-oriented design.

Keywords. Design patterns, object-oriented design, object-oriented software development, application frameworks, class libraries, reusability

1 Introduction

One of the principal goals of object-oriented software development is to improve the reusability of software components. Increased reusability of software is considered as crucial technical precondition to improve the overall software quality and reduce production and maintenance costs.

Reuse of Single Components. Conventional function-/procedure libraries can be viewed as sets of small building blocks that might be compared to elemenatry components such as screws and bolts in the real world. Libraries that offer more complex functions and procedures based on the 'call-back' principle (for example, some libraries for GUI programming belong to that category) turned out to be often too inflexible and too difficult to use.

Module-oriented languages such as Ada and Modula-2 allow to express the concept of *Abstract Data Types* (ADTs). Unfortunatley, it is almost impossible to construct such modules in a way so that they meet all future requirements no matter in which project the modules will be reused. Typically, a small 'delta' has to be changed so that a module can be reused in software projects other than the one it was originally designed for.

Such variants of the original module are not compatible any more. Furthermore, chances are high that these delta changes in the source code imply errors.

As a consequence, module-oriented programming could not lead to the awaited breakthrough in software reusability. Only pretty simple modular components such as linked lists, sets, hash tables, etc. are reusable in numerous projects without modifications.

Since object-oriented languages support that *programming by difference* this problem is alleviated: ADTs can often be adapted without touching the source code of the original ADT. The adapted ADTs are compatible to the original one.

Concepts offered by object-oriented programming languages are used in many projects for the sole purpose of producing reusable *single* components.

Reuse of Architectures. The concepts inheritance and dynamic binding are sufficient to construct so-called *frameworks*, that is, reusable semi-finished architectures for various application domains. Such frameworks mean a real breakthrough in software reusability: not only single building blocks but whole software (sub-)systems including their design can be reused.

Though most of the matured framworks exist for the GUI domain (such as MacApp [1], and ET++ [4, 9, 10]) the framework concept can be applied to any application domain. The term application framework is often used for frameworks which constitute a generic application for a domain area. We use the term framework for both application framework and 'small' framework that might consist of a few components and be part of an application framework. In cases where a distinction is necessary the term application framework is explicitly used.

2 The Role of Design Patterns

As the design pattern approach in the realm of object-oriented software development just emerged recently there is no consensus on the term 'design pattern'. Various design pattern approaches differ in their purpose. In general, we might discern between design pattern approaches that focus on the design of single components or a small group of components ignoring the framework concept. Other approaches, such as the one of Erich Gamma et al. [5, 6] and the meta pattern approach presented in this paper pursue the goal of supporting the design of frameworks.

Design Patterns and Frameworks. Experts in the field of object-oriented technology have intuitively recommended that programmers who want to learn how to develop frameworks should start learning basic object-oriented concepts first and then proceed to take a close look at various frameworks. The major disadvantage of that approach is the enormous effort required. Since the first step of this 'method' of learning is to use a framework, programmers have to learn all the details of a framework. Often this involves learning an additional programming language. Poor documentation of available frameworks makes the second step—studying implementation details—even more painful and time consuming. Abstracting *design patterns* which have been obscured by many implementation details requires an in-depth look at a framework. Sometimes it becomes even impossible to understand particular design decisions without any hints.

The main purpose of the framework-centered design pattern approaches is to describe the design of a framework and its individual classes without revealing the implementation details. Such abstract design descriptions constitute an indispensable vehicle for communicating matured designs in an efficient way. As a result, design patterns can help to

- understand the specific implementation details of a framework, as design patterns constitute a 'road-map' for doing this
- construct new frameworks which incorporate matured and proven designs

A pioneering work was accomplished by Erich Gamma in his doctoral thesis [4] which describes design patterns in the ET++ application framework [9, 10]. The work presented in this paper has been inspired by the OOPSLA'91 - OOPSLA'93 workshops and Erich Gamma's design pattern descriptions.

Design Patterns and OOAD Methodologies. State-of-the-art OOAD methodologies such as the Object Model Technique (OMT) [8], Booch's method [2] and the Responsibility-Driven-Approach [12] are limited in that the development and adaptation of frameworks are not primarily addressed. This is true for all of the numerous OOAD methodologies. Though there are significant differences in the applied notation, the methodologies themselves are not radically different. Rumbaugh et al. [8] express this in the following way: 'All of the object-oriented methodologies, including ours, have much in common, and should be contrasted more with non-object-oriented methodologies than with each other.'

OOAD methodologies assist in the development of a well-structured object-oriented system. Frameworks have to evolve from this initial framework design. Design patterns can support this architecture evolution. In that sense, OOAD methodologies are complemented by design pattern approaches.

3 Hot Spots of Frameworks

Application frameworks consist of ready-to-use and semi-finished building blocks The overall architecture (= the composition and interaction of building blocks) is predefined as well. Producing specific applications usually means to adjust building blocks to specific needs by overriding some methods in subclasses.

In general, an application framework standardizes applications for a specific domain. At the same time some aspects cannot be anticipated. These parts of an application framework have to be generic so that they can easily be adapted to specific needs. Figure 1 shows schematically this property of an application framework with the flexible parts in gray color.

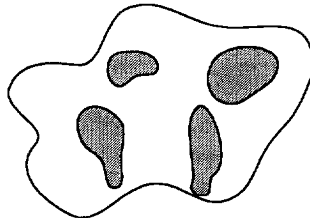

Fig. 1. An application framework with flexible hot spots

The difficulty of 'good' object-oriented design is to identify the *hot spots* of an application framework, i.e., those aspects of an application domain that have to be kept flexible. Remember how Wirfs-Brock and Johnson outline this problem: 'Good frameworks are usually the result of many design iterations and a lot of hard work. ....

Lack of generality in a framework shows up when it is used to build applications, so its weaknesses cannot be found until after it is designed and reused' [11]. We consider a framework to have the quality attribute 'well designed' if it provides adequate hot spots for adaptations.

Primarily, *domain-specific knowledge is required* to find those hot spots. Only domain analysis can help to acquire this knowledge. Design patterns are useless during this domain analysis. Patterns can only outline how to design and implement frameworks that adhere to these hot spots, that is, frameworks that are adaptable where required.

4 Meta Patterns

Framework-centered design pattern approaches, such as the design pattern catalog [5, 6] attempt to pick out *framework examples* 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. We use the term framework example design patterns for those design patterns. Framework example design patterns mainly differ in the semantic aspect of the hot spot that is kept flexible.

Since experienced object-oriented designers collected these framework example design patterns we consider these catalogs as useful means to construct new frameworks. Furthermore, framework example design patterns typically include implementation hints. Nevertheless, we think that a more advanced abstraction is helpful, for example, in order to actively support the design pattern idea in the realm of tools.

We introduce the term *meta patterns* for a set of design patterns that describes how to construct frameworks independent of a specific domain. Actually, it is pretty straight-forward to construct frameworks by combining the basic object-oriented concepts. Thus these meta patterns turn out to be an elegant and powerful approach that can be applied to categorize and describe any framework example design pattern on a *meta-level*. So meta patterns do not replace state-of-the-art design pattern approaches but complement them.

The Object Model Notation proposed by Rumbaugh et al. [8] is used in order to depict class and object diagrams.

4.1 Class/Object Interface and Interaction Meta Patterns

So-called *template and hook methods* represent the meta patterns required to design frameworks consisting of single classes or groups of classes together with their interactions. The terms 'template method' and 'hook method' are commonly used by various authors such as [5, 12].

Note that the term 'template method' or simply 'template' must not be mixed up with the C++ construct 'template' which has a completely different meaning. Furthermore, we assume that all methods are defined as dynamically bound ones.

The *narrow inheritance interface principle* [10] as fundamental design guideline is strongly related to the distinction between hook and template methods. General considerations and specific examples illustrate how to combine these two meta patterns in order to develop well designed frameworks.

Flexibility Within One Class. Let us consider an example (see Figure 2) assuming that a class B offers three methods M1(), M2() and M3(). In this example

M1() constitutes the template method based on the hook methods M2() and M3(). For method M2() only the method interface (name and parameters) can be defined, not an implementation. The term *abstract method* is often used in such a case. Classes that contain abstract methods are termed abstract classes. This is expressed graphically by writing the method name(s) and class name in *italic style*. Method M3() is assumed to provide a meaningful default implementation.

In the figures we use the C++ notation ClassName:: in order to express that a method belongs to a class.

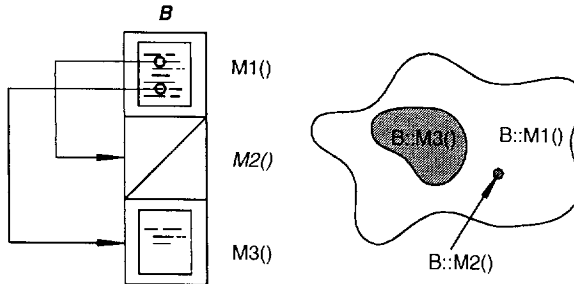

Fig. 2. A template method calling its hook methods

Subclass B1 adapts the template method M1() by overriding M2(). Thus the hot spot M2() is filled. The template method M1() of class B is adapted without changing its source code as illustrated in Figure 3.

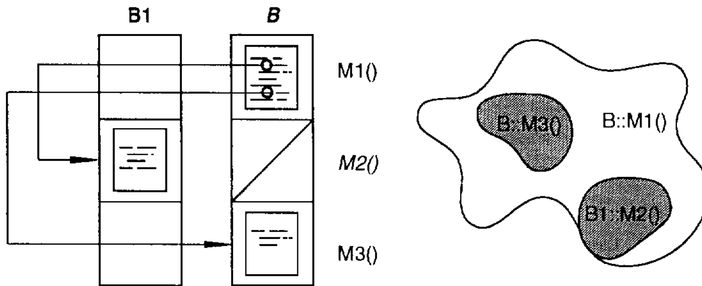

Fig. 3. Adaptation of template method M1() by overriding B::M2()

Of course, method M3() can also be overridden in a subclass of B in order to adapt this hot spot of template method M1().

Flexibility Across Class Borders. In general, methods of a class B establish a *contract*—subclasses can only modify method implementations or add new methods. Either methods of class B itself (as shown in the example above) or of any other class can be based on the contract of B ('based on the contract of B' means that variables of static type B are used and messages corresponding to the contract of B are sent to these B objects). The following example illustrates the case that another class A is based on the contract of B. Due to polymorphism, other components of the application framework that are based on B, work with instances of any subclass of B. What actually happens at run-time depends on the object's run-time type, i.e., how the

particular hook methods are implemented in the corresponding subclasses of B. The term abstract coupling is used to express that.

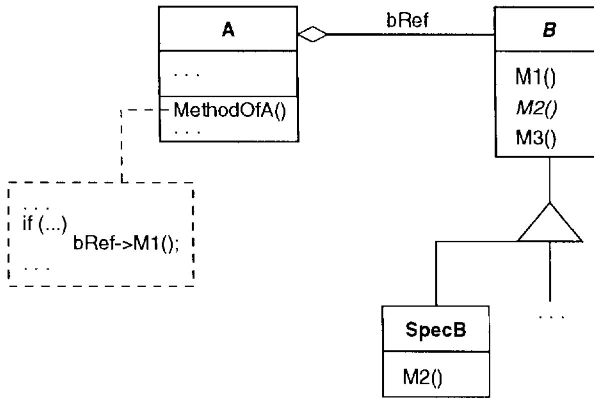

Fig. 4. Abstract coupling based on template and hook methods

In the example in Figure 4 objects of class A maintain a reference to objects of class B by means of the instance variable bRef.

Ideally, MethodOfA() is adapted to specific needs by only overriding B::M2() in a subclass of B. The instance variable bRef then has to refer to the appropriate instance of a subclass of B.

It depends on the point of view what constitutes a template method and a hook method. 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, MethodOfA() is a template method using M1() as hook method. In the realm of class B method M1() is a template method.

Template methods call at least one other method. Hook methods can be abstract methods, regular methods that call no other methods or again template methods.

Narrow Inheritance Interface Principle. Weinand et al. [10] explain the purpose of the narrow inheritance interface principle employed in frameworks: 'Behavior that is spread over several methods in a class should be based on a minimal set of dynamically bound methods which have to be overridden. This allows clients deriving subclasses from an existing class to override just a few methods in order to adapt its behavior. Not adhering to this narrow inheritance interface principle often means that too many methods have to be overridden, resulting in ugly and bulky code.'

For example, the template method B::M1() (see Figure 2 and Figure 3) adheres to the narrow inheritance principle if clients that derive a subclass from B only have to override B::M2(), and maybe also B::M3() in order to adjust the behavior of B::M1(). The same should be true for adjusting A::MethodOfA().

Unfortunateley, designing classes with narrow inheritance interfaces comprises conflicting goals: a good design of class interfaces in the realm of frameworks should find the optimum balance between flexibility and the effort to adapt classes. A class that provides powerful template methods which are only based on a few hook methods implies a minimal adaptation effort: behavior is adapted by overriding these hook

methods in subclasses. But this could sacrifice the flexibility of a class. If the hook methods are not sufficient to modify the template method's behavior, the class becomes inflexible, implying that the whole template method has to be overridden. Thus template methods have the potential of reducing adaptation effort without sacrificing flexibility. They add the danger of making classes too rigid, which again drastically increases the adaptation effort.

As a consequence, classes in a mature framework should contain powerful, yet flexible template methods. Clients ideally only have to override abstract hook methods in order to adapt a framework to their specific needs. The design of the corresponding class interfaces typically requires many iterations (caused by the usage of classes in specific situations). During such iterations template methods appear to be too rigid so that more hook methods have to be integrated. Of course, it is also possible that the usage of classes provides insights how to define additional template methods, if clients have to implement similar control flow again and again.

It seems impossible to describe general guidelines for getting template methods right the first time. As already mentioned in the beginning, class interface design depends very much on the various domains for which frameworks are developed. Examples can only illustrate how to do it right in specific situations. Above all, it is important to get the basic idea of template methods (they essentially implement the framework concept) and to see the conflicting goals of class interface design which result from the narrow inheritance interface principle.

The structure of template methods sometimes depends on the way objects are composed. This issue is discussed in 4.3.

4.2 Class/Object Composition Meta Patterns

Sometimes, white spots (template methods) and gray spots (hook methods) are unified in one class. In many situations it is more adequate to put white spots and hot spots into separate classes. In that case, the class that contains the hook method(s) can be considered as *hook class* of the class that contains the corresponding template method(s). We call the class which contains the template method(s) *template class*. In other words, a hook class *parameterizes* the corresponding template class.

Analogous to template and hook methods it depends on the particular situation which class is a template and which one a hook class.

The following attributes are relevant in order to describe the way how the corresponding objects of a template class and a hook class can be composed:

- Can an object of a template class refer to exactly one other object or to any number of objects of the hook class?
- Is the template class a descendant of the hook class?

(Details why these attributes are relevant are discussed in [7].) In order to allow message sending between objects of template and hook classes, there has to exist a reference relationship between the corresponding objects. Though there are several ways to establish such a reference relationship in the realm of composition meta patterns we consider only the most important one, i.e., the establishment of a reference relationship via an instance variable.

Based on a combination of the composition attribute values the composition meta patterns depicted in Figure 5 result.

The Unification meta pattern represents the special case that template and hook classes are unified in one class. For example, class B in Figure 2 is based on the

Unification meta pattern with template method and corresponding hook methods in one class.

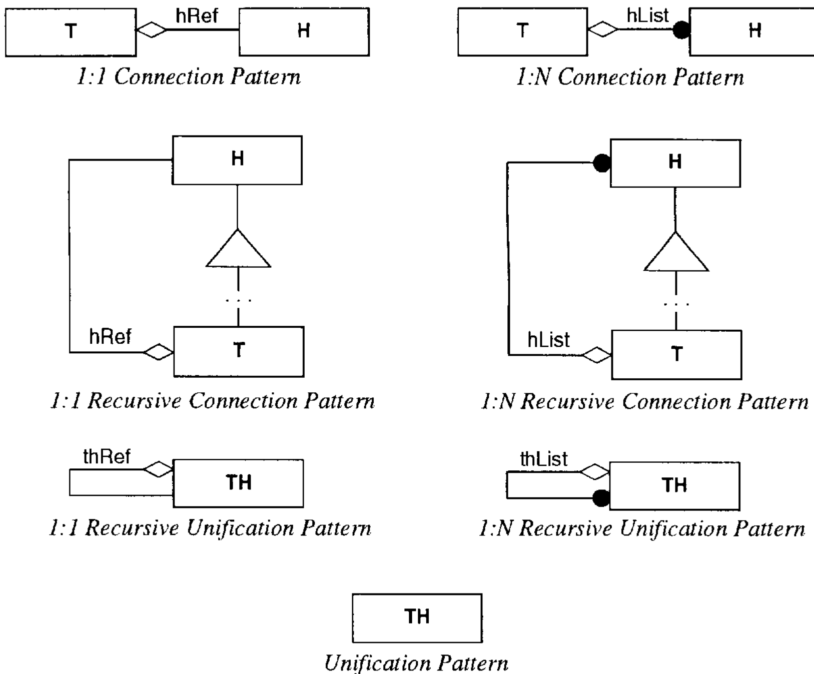

Fig. 5. Composition meta patterns

4.3 Impact of Composition Meta Patterns on Template Methods

All composition meta patterns except for the 1:1 Connection and the Unification pattern imply a typical structure of the template methods. The various structures of template methods result directly from the composition attributes. We pick out the 1:N Connection pattern in order to illustrate this.

In the recursive connection patterns template methods and hook methods have typically the same name, for example, TH(). The template method T::TH() overrides the hook method H::TH(). Figure 6 shows the structure of TH() in the 1:N Recursive Connection pattern.

```
... T::TH(...) {
        . . .
        for each hookObject: <H *> in hList
                hookObject->TH(...);
        . . .
}
```

Fig. 6. Structure of template methods in the 1:N Recursive Connection pattern

Since hookObject is of static type 'pointer to H', objects of any subclass of H, that is, also recursively T objects can be handled. This allows to build up directed graphs of T objects and H objects. Figure 7 shows an example of such a graph.

Note that leaves of the tree can also be T objects: T objects may refer to zero or more H objects. On the other hand, H objects can never be (sub)roots of the tree.

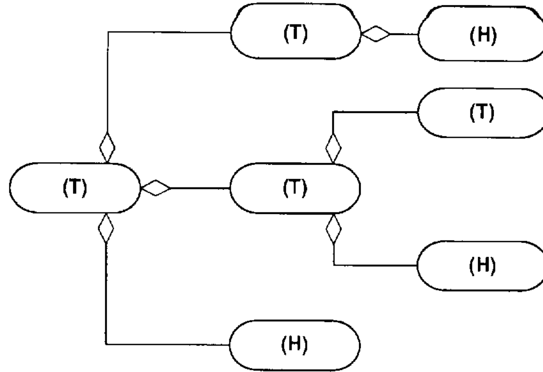

Fig. 7. Tree-hierarchy of T objects and H objects

What is the purpose of the template method in the realm of the 1:N Recursive Connection pattern?

Due to the structure of the template method shown in Figure 6 messages are automatically forwarded along the objects in a directed graph. In general, 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.

The structure of the other composition meta patterns and guidelines when to choose them are discussed in detail in [7]. Composition meta patterns especially differ in the offered degree of flexibility of the hot spots. For example, choosing the Unification meta pattern implies the following: in order to change the template method by providing a different hook method a subclass of the unified template-hook class TH has to be defined. Thus changes of the template method cannot occur dynamically at runtime.

More flexibility is provided if template and hook classes are separated. This means that 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 be done at runtime.

5 Application of Meta Patterns

Design pattern catalogs list framework *examples*, i.e., describe the design of specific frameworks on an abstraction level higher than the underlying programming language. Since the examples are carefully chosen so that they are not too domain-specific these

framework example design patterns can be reused in other frameworks. However, they provide no means to capture the design independent of a more or less specific framework example.

Meta patterns presented in this paper are suited for the design documentation of any (application) framework. Below we outline the principal idea how meta patterns can be 'attached' to frameworks. In this way meta patterns express how the required flexibility—represented by the hot spots—is gained in a particular framework. These design hints are often necessary in order to adapt frameworks. Programmers who develop new frameworks might benefit from studying the design of existing frameworks and applying it to frameworks under development.

We exemplify the attachement of meta patterns to framework examples for the 1:N Recursive Connection meta pattern and the 1:1 Connection meta pattern.

1:N Recursive Connection Meta Pattern. The 1:N Recursive Connection pattern is, for example, applied in a small framework consisting of the classes Folder and Item. A Folder object can manage any number of Item objects. Figure 8 illustrates by means of arrows how the components of the 1:N Recursive Connection meta pattern correspond to the classes and methods of this small framework.

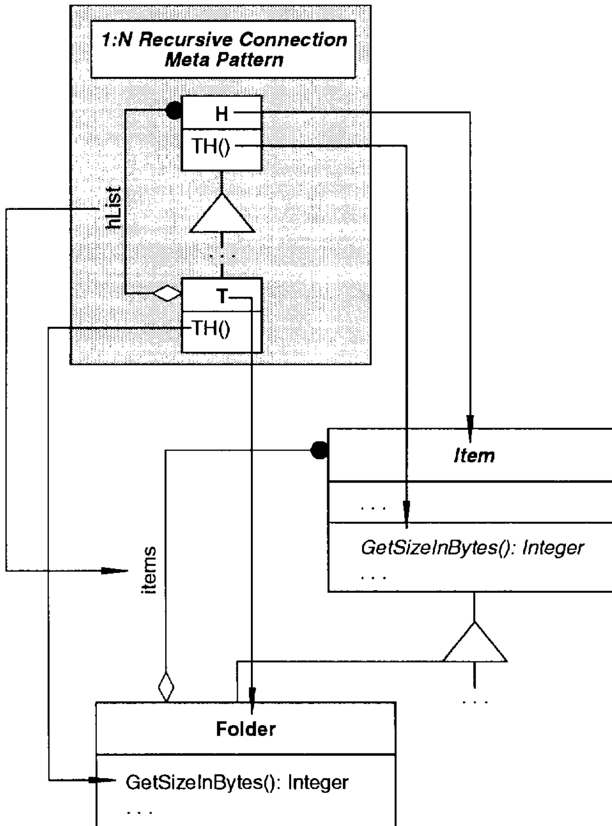

Fig. 8. Attaching the 1:N Recursive Connection meta pattern to a sample framework

The semantic aspect of the hot spot in that sample framework is the way Item objects calculate their size.

Due to the characteristics of the 1:N Recursive Connection pattern Folder objects can be treated as single Item objects so that hierarchies can be composed. Requests such as GetSizeInBytes are automatically forwarded within an object hierarchy (assuming that the template method GetSizeInBytes adheres to the typical structure of template methods in the 1:N Recursive Connection pattern). Furthermore, the behavior that is kept flexible by means of this pattern, that is, the size calculation, can be adapted at runtime—Folder objects calculate their size correctly if the contained items are changed.

1:1 Connection Meta Pattern. For example, in the domain of reservation systems the initial *hot-spot analysis* might reveal that the rate calculation has to become one of the hot spots of an application framework for that domain. If the way how rental rates are calculated for a rental item has to be kept flexible the 1:1 Connection meta pattern could be chosen in the framework design as shown in Figure 9.

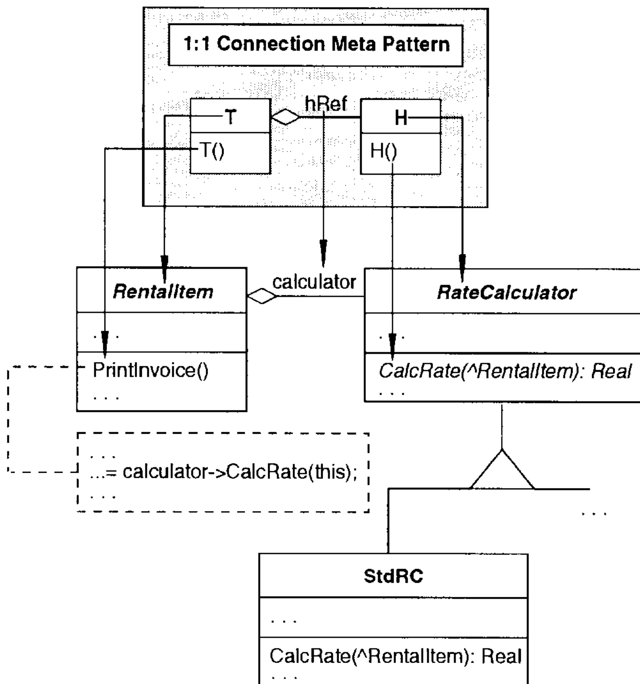

Fig. 9. Attaching the 1:1 Connection meta pattern to a sample framework

So reservation systems built by adapting that framework have to specify the rate calculation in a subclass of RateCalculator. For example, objects of a class HotelRoom (as subclass of RentalItem) in a hotel reservation system will require a different RateCalculator object than instances of a class Vehicle in a rental car reservation system. Due to the characteristics of the 1:1 Connection meta pattern, the rate calculation behavior can be switched at runtime by coupling a RentalItem object with various RateCalculator objects—one at a time.

6 Summarizing Remarks and Outlook

The seven composition meta patterns repeatedly occur in frameworks. Each framework uses, of course, specific names for the template and hook classes and the corresponding methods, depending on the semantics of the hot spots and the white spots. The core characteristics of the composition meta patterns are independent of their particular application.

Thus, a hypertext design browser could be based on the meta pattern descriptions to provide a means to browse through numerous examples where a composition meta pattern is combined with domain-specific template and hook methods.

The fact that *myriads of meta pattern annotations* are possible in (application) frameworks might be considered as disadvantage: almost each method calls other methods and thus becomes a template method that is based on hook methods. Meta patterns could be attached to all calls where a particular method is the template method and the invoked methods are hook methods. Persons that know a framework well enough, especially the developers of a framework can select those aspects that should be put into a meta pattern browser.

Meta patterns are useful when they are attached to already matured frameworks. It does not make sense to define meta pattern browsers for frameworks that are in their early development stages, i.e., where it is still not clear whether template and hook classes are defined and implemented according to the needs of the framework domain.

Meta pattern browsers for matured (application) frameworks can be viewed as advanced design pattern catalogs. Some aspects of a specific framework might be pretty domain-independent so that this design can be applied in the development of new frameworks. In these cases meta pattern browsers 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 meta pattern browsers.

In addition to design pattern catalogs meta pattern browsers can instrument any domain-specific framework and document its design. The aspect that meta pattern browsers allow an efficient design documentation of frameworks can help in adapting the hot spots of a framework to specific needs.

Future research based on a prototype implementation of a meta pattern browser will reveal the suitability of a design documentation based on meta patterns for the adaptation and development of frameworks.

References

1. Apple Computer: MacApp II Programmer's Guide; 1989.

2. Booch G.: Object-Oriented Design; Redwood City, CA, Benjamin/Cummings, 1991.

3. Coad P.: Object-Oriented Patterns; in Communications of the ACM, Vol. 33, No. 9, Sept. 1992.

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

5. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Abstraction and Reuse of Object-Oriented Design; in ECOOP'93 Conference Proceedings, Springer Verlag, 1993.
6. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns—Microarchitecturs for Reusable Object-Oriented Software; Addison-Wesley, 1994.
7. Pree W.: Design Patterns for Object-Oriented Software Development; (preliminary title) to be published by Addison-Wesley/ACM Press, 1994.
8. Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: Object-Oriented Modeling and Design; Prentice Hall, Englewood Cliffs, New Jersey, 1991.
9. 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.
10. 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.
11. Wirfs-Brock R.J., Johnson R.E.: Surveying Current Research in Object-Oriented Design; in Communications of the ACM, Vol. 33, No. 9, 1990.
12. Wirfs-Brock R., Wilkerson B., Wiener L.: Designing Object-Oriented Software; Prentice Hall, Englewood Cliffs, New Jersey, 1990.