Essential Framework Design Patterns

Wolfgang Pree

University of Constance
D-78434 Constance, Germany
Voice: +49.7531.88.44.33; Fax: +49.7531.88.35.77
E-mail: pree@informatik.uni-konstanz.de

Abstract. Most excellent object-oriented frameworks are still the product of a more or less chaotic development process, typically carried out in the realm of research-like settings. This contribution first discusses the few essential framework construction principles. Frameworks represent a generic solution for a particular domain and enable the exploitation of the full potential of object-oriented software construction: Not only source code and single components but also architectural design is reused. Adaptation takes place at points of predefined refinement that we call *hot spots*. If these hot spots are identified explicitly in the requirements phase, they can be combined with the essential construction patterns to form domain-specific design patterns. Such a hot-spot-driven framework design can contribute to a more systematic development process.

Key words: design patterns, frameworks, object-oriented design, object-oriented software development, software reusability

1 Framework types

Frameworks are well suited for domains where numerous similar applications are built from scratch again and again. A framework defines a *high-level language* with which applications within a domain are created through *specialization* (= adaptation). Specialization takes place at points of predefined refinement that we call *hot spots*. We consider a framework to have the quality attribute *well designed* if it provides adequate hot spots for adaptations. For example, Lewis et al. (1995) present various high-quality frameworks.

Hot spots in white-box frameworks

White-box frameworks consist of several incomplete classes, that is, classes that contain methods without meaningful default implementations. Class `A` in the sample framework class hierarchy depicted in Figure 1 illustrates this characteristic of a white-box framework. The abstract method of class `A` that has to be overridden in a subclass is drawn in gray. The abstract methods form the hot spots in this type of framework.

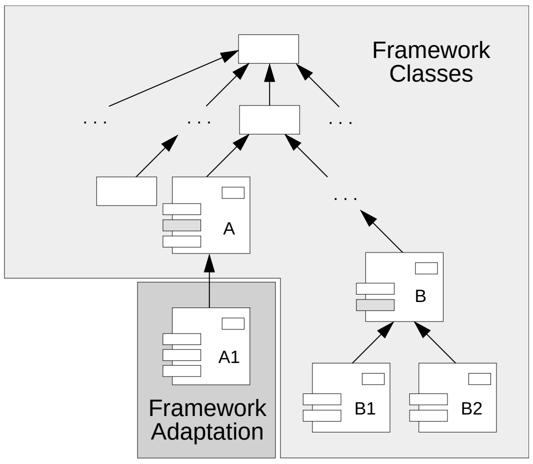

Figure 1 Sample framework class hierarchy (from Pree, 1996)

Programmers modify the behavior of white-box frameworks by applying inheritance to override methods in subclasses of framework classes. The necessity to override methods implies that programmers have to understand the framework's design and implementation, at least to a certain degree of detail.

Hot spots in black-box frameworks

Black-box frameworks offer ready-made components for adaptations. Modifications are done by *composition*, not by programming. Hot spots also correspond to the overridden method(s), though the one who adapts the framework only deals with the components as a whole.

In the framework class hierarchy in Figure 1, class `B` already has two subclasses `B1` and `B2` that provide default implementations of `B`'s abstract method. Supposed that the framework components interact as depicted in Figure 2(a). (The lines in Figure 2 schematically represent the interactions between the components.) A programmer adapts this framework, for example, by instantiating classes `A1` and `B2` and plugging in the corresponding objects (see Figure 2(b)). In the case of class B, the framework provides ready-to-use subclasses; in the case of class A the programmer has to subclass A first.

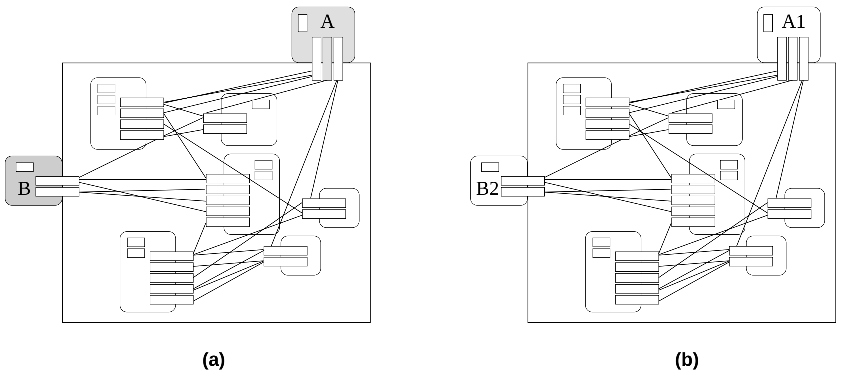

Figure 2 Framework (a) before and (b) after specialization by composition.

Available frameworks are neither pure white-box nor pure black-box frameworks. If the framework is heavily reused, numerous specializations will suggest which black-box defaults could be offered instead of just providing a white-box interface. So frameworks will evolve more and more into black-box frameworks when they mature.

2 Flexibility through hooks

Methods in a class can be categorized into socalled hook and template methods: Hook methods can be viewed as place holders or flexible hot spots that are invoked by more complex methods. These complex methods are usually termed template methods[1] (Wirfs-Brock *et al.*, 1990; Gamma *et al.*, 1995; Pree, 1995). Template methods define abstract behavior or generic flow of control or the interaction between objects. The basic idea of hook methods is that overriding hooks through inheritance allows changes of an object's behavior without having to touch the source code of the corresponding class. Figure 3 exemplifies this concept which is tightly coupled to constructs in common object-oriented languages. Method t() of class A is the template method which invokes a hook method h(), as shown in Figure 3(a). The hook method is an abstract one and provides an empty default implementation. In Figure 3(b) the hook method is overridden in a subclass A1.

1 Template methods must not be confused with the C++ template construct, which has a completely different meaning.

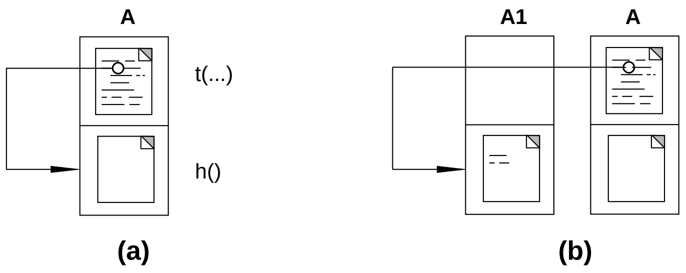

Figure 3 (a) Template and hook methods and (b) hook overriding.

Let us define the class that contains the hook method under consideration as *hook class* H and the class that contains the template method as *template class* T. A hook class quasi parameterizes the template class. Note that this is a context-dependent distinction regardless of the complexity of these two kinds of classes. As a consequence, the essential set of flexibility construction principles can be derived from considering all possible combinations between these two kinds of classes. As template and hook classes can have any complexity, the construction principles discussed below scale up. So the domain-specific semantics of template and hook classes fade out to show the clear picture of how to achieve flexibility in frameworks.

2.1 Unification versus separation patterns

In case the template and hook classes are unified in one class, called TH in Figure 4(a), adaptations can only be done by inheritance. Thus adaptations require an application restart.

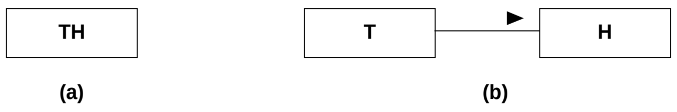

Figure 4 (a) Unification and (b) separation of template and hook classes.

Separating template and hook classes is equal to (abstractly) coupling objects of these classes so that the behavior of a T object can be modified by composition, that is, by plugging in specific H objects.

The directed association between T and H expresses that a T object refers to an H object. Such an association becomes necessary as a T object has to send messages to the associated H object(s) in order to invoke the hook methods. Usually an instance variable in T maintains such a relation. Other possibilities are global variables or temporary relations by passing object references via method parameters. As the actual coupling between T and H objects is an irrelevant implementation detail, this issue is not discussed in further detail. The same is true for the semantics expressed by an association. For example, whether the object association indicates a *uses* or *is part of* relation depends on the specific context and need not be distinguished in the realm of these core construction principles.

A separation of template and hook classes also forms the precondition of *run-time adaptations*, that is, subclasses of H are defined, instantiated and plugged into T objects while an application is running. Gamma et al. (1995) and Pree (1996) discuss some useful examples.

2.2 Recursive combination patterns

The template class can also be a descendant of the hook class (see Figure 5(a)). In the degenerated version, template and hook classes are unified (see Figure 5(b)). The recursive compositions have in common that they allow building up directed graphs of interconnected objects. Furthermore, a certain structure of the template methods, which is typical for these compositions, guarantees the forwarding of messages in the object graphs.

The difference between the simple separation of template and hook classes and the more sophisticated recursive separation is that the playground of adaptations through composition is enlarged. Instead of simply plugging two objects together in a straightforward manner, whole directed graphs of objects can be composed. The implications are discussed in detail in Pree (1995, 1996).

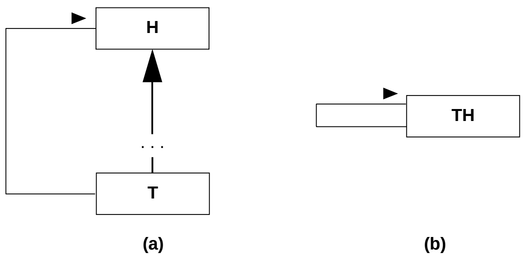

Figure 5 Recursive combinations of template and hook classes.

2.3 Hooks as name designators of GoF pattern catalog entries

Below we assume that the reader is familiar with the patterns in the pioneering Gang-of-Four catalog (Gamma et al., 1995). Numerous entries in the GoF catalog represent small frameworks, that is, frameworks consisting of a few classes, that apply the few essential construction patterns in various more or less domain-independent situations. So these catalog entries are helpful when designing frameworks, as they illustrate typical hook semantics. In general, the names of the catalog entries are closely related to the semantic aspects that are kept flexible by hooks.

Patterns based on template-hook separation

Many of the framework-centered catalog entries rely on a separation of template and hook classes (see Figure 4(b)). Two catalog patterns, Template Method and Bridge, describe this construction principle. The following catalog patterns are based on abstract coupling: Abstract Factory, Builder, Command, Interpreter, Observer, Prototype, State and Strategy. Note that the names of these catalog patterns correspond to the semantic aspect which is kept flexible in a particular pattern. This semantic aspect again is reflected in the name of the particular hook method or class. For example, in the Command pattern "when and how a request is fulfilled" (Gamma et al., 1995) represents the hot spot semantics. The names of the hook method (Execute()) and hook class (Command) reflect this and determine the name of the overall pattern catalog entry.

Patterns based on recursive compositions

The catalog entries Composite (see Figure 5(a) with a 1: many relationship between T and H), Decorator (see Figure 5(a) with a 1:1 relationship between T and H) and Chain-of-Responsibility (see Figure 5(b)) correspond to the recursive template-hook combinations.

3 How to find domain-specific patterns

Hot spot identification in the early phases (eg, in the realm of requirements analysis) should become an explicit activity in the development process. There are two reasons for this: Design patterns, presented in a catalog-like form, mix construction principles and domain specific semantics as sketched above. Of course, it does not help much, to just split the semantics out of the design patterns and leave framework designers alone with bare-bone construction principles. Instead, these construction principles have to be combined with the semantics of the domain for which a framework has to be developed. Hot spot identification provides this information. Figure 6 outlines the synergy effect of essential construction principles paired with domain-specific hot spots. The result is design patterns tailored to the particular domain.

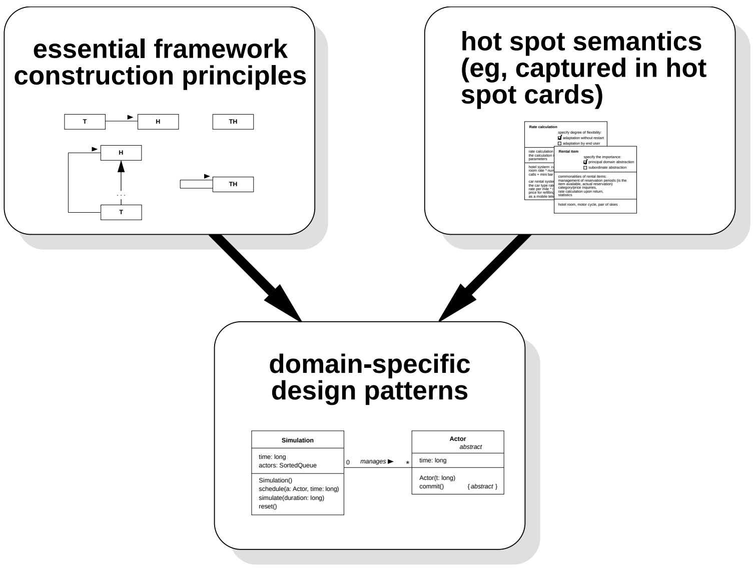

Figure 6 Essential construction principles + hot spots = domain-specific design patterns

Hot spot identification can be supported by hot spot cards, a communication vehicle between domain experts and software developers. Pree (1996) presents the concept of hot spot cards and detailed case studies where they are applied.

A further reason why explicit hot spot identification helps, can be derived from the following observations of influencing factors in real-world framework development: One seldom has two or more similar systems at hand that can be studied regarding their commonalities. Typically, one too specific system forms the basis of framework development. Furthermore, commonalities should by far outweigh the flexible aspects of a framework. If there are not significantly more standardized (= frozen) spots than hot spots in a framework, the core benefit of framework technology, that is, having a widely standardized architecture, diminishes. As a consequence, focusing on hot spots is likely to be more successful than trying to find commonalities.

4 Outlook

Above we discussed technical aspects of framework development by presenting the fundamental framework design patterns. But organizational measures are at least equally important to be successful as framework development requires a radical departure from today’s project culture. Goldberg and Rubin (1995) present these aspects in detail.

Overall framework development does not result in a short-term profit. On the contrary, frameworks represent an investment that pays off in the long term. But we view frameworks as the long-term players towards reaching the goal of developing software with a building-block approach. Though the state of the art still needs profound refinement, many currently existing frameworks corroborate that frameworks will be the enabling technology in many areas of software development.

A word of advice for those who have not worked with frameworks so far: No methodology or design technique will help avoid this painful learning process. Toy around with some of the available large-scale frameworks and get a better understanding of the technology by first reusing frameworks before jumping into framework development.

5 References

Gamma E., Helm R., Johnson R. and Vlissides J. (1995). *Design Patterns—Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley

Goldberg A., Rubin K. (1995). *Succeeding with Objects—Decision Frameworks for Project Management*. Reading, Massachusetts: Addison-Wesley

Lewis T. et al. (1995). *Object-Oriented Application Frameworks*. Greenwich, CT: Manning Publications/Prentice Hall.

Pree W. (1995). *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley/ACM Press

Pree W. (1996). *Framework Patterns*. New York City: SIGS Books

Wirfs-Brock R. and Johnson R. (1990). Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, **33**(9)