Hot-Spot-Driven Framework Development

Wolfgang Pree

Software Research Lab
University of Constance
D-78457 Constance, Germany
Voice: +49-7531-88-44 33; Fax: +49-7531-88-35 77
E-mail: pree@acm.org

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. Overall, flexibility has to be injected into a framework in appropriate doses[1]. Framework adaptation takes place at points of predefined refinement that we call *hot spots*. As the quality of a framework depends directly on the appropriateness of hot spots, hot spot identification has to become an explicit activity in the framework development process. Means for documenting and communicating hot spots between domain experts and software engineers become crucial.

This contribution first discusses the few essential framework construction principles, that is, how to keep object-oriented architectures flexible for adaptations. We introduce hot spot cards as means to capture flexibility requirements, and illustrate how to apply them in combination with the essential framework construction principles. The presented heuristics form a hot-spot-driven framework design process which leads to a more systematic framework construction with fewer (re)design iterations.

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

[1] Parnas (1976) discusses this issue in general, using the terms software family and commonality/ variability analysis

1 Hot spots in white-box and black-box frameworks

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* (Pree, 1995, 1996, 1997). Specialization is accomplished through (black-box) composition or (white-box) inheritance as explained below. 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) as well as many contributions in this book present various high-quality frameworks.

Hot spots based on inheritance or interfaces

The framework attributes white-box and black-box categorize its hot spots. A framework is neither pure black-box nor pure white-box. The term white-box framework is used synomymously for white-box aspect of a framework. A white-box framework comprises 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 method forms the hot spot in this case.

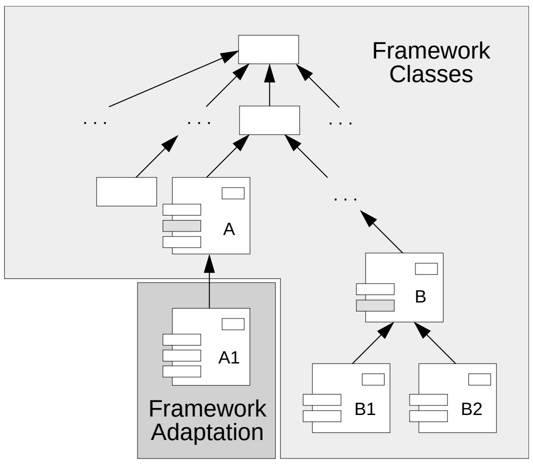


Figure 1 Sample framework class hierarchy (from Pree, 1996)

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.

Java interfaces represent pure abstract classes consisting only of abstract methods. Interfaces allow the separation of class and type hierarchies. In the sample class hierarchy in Figure 1, the abstract class A could be defined as interface. In the original design, only instances of subclasses of A were type-compatible to A. In case of defining an interface A, an instance of any class in the class hierachy that implements the interface is of type A. Of course, the white-box characteristic of a framework does not change in case of using interfaces. If no classes implement an interface in an adequate way, the programmer who adapts the framework also has to understand partially the framework's design and implementation.

Hot spots based on composition

Black-box frameworks offer ready-made components for adaptations. Modifications are done by simple *composition*, not by tedious inheritance 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. Let us assume 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 case of class B, the framework provides ready-to-use subclasses; in case of class A the programmer has to subclass A first.

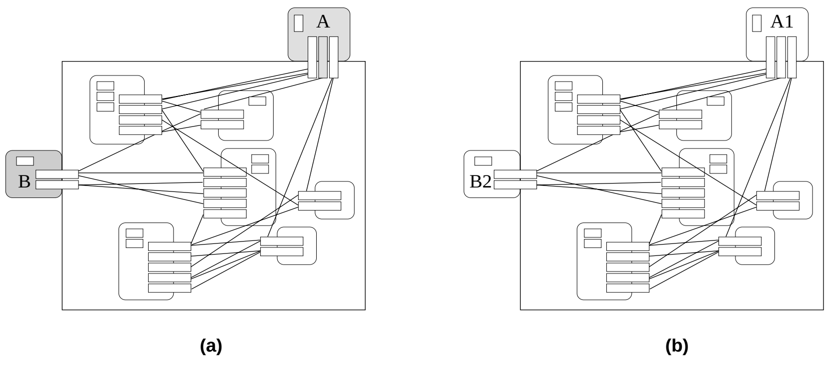


Figure 2 Framework (a) before and (b) after specialization.

Remeber that 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 Hook methods as elementary building blocks of hot spots

This section discusses the essential framework construction patterns which form the basis of the hot-spot-driven framework design process (see Section 3).

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[2] (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

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

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.

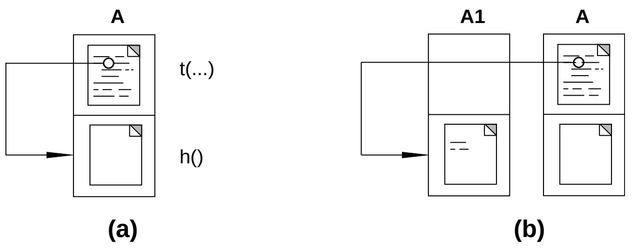


Figure 3 (a) Template and hook methods and (b) hook overriding.

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

If 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.

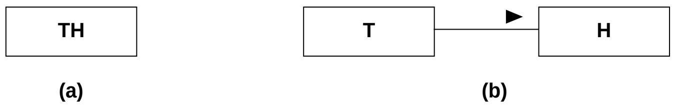


Figure 4 (a) Unification and (b) separation of template and hook classes.

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. Also note that T and H just represent types. So H, for example, could be a Java interface as well.

A separation of template and hook classes 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 various 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)). T as a subtype of H can manage another T instance. Thus these patterns are termed recursive compositions. 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 as in the Separation pattern, whole directed graphs of objects can be composed. The characteristics and implications are discussed in detail in Pree (1995, 1996, 1997).

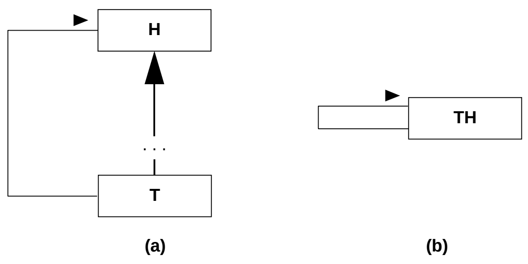


Figure 5 Recursive combinations of template and hook classes.

2.3 Hooks as name designators of GoF pattern catalog entries

In this section we assume that the reader is familiar with the patterns in the pioneering Gang-of-Four pattern 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 the Unification and Separation construction principle. The following catalog patterns rely on the Separation pattern: 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 Hot-spot-driven development process

The pain of designing a framework is already described by Wirfs-Brock and Johnson (1990): "Good frameworks are usually the result of many design iterations and a lot of hard work". So don't expect a panacea. No framework will be ideal from the beginning. More realistically, there should be means to reduce the number of design iterations. Thus the term *framework development* expresses both the initial design and the evolution of a framework.

As the quality of a framework is directly related to the flexibility required in a domain, explicit identification of domain-specific hot spots can indeed help. Figure 6 gives an overview of the hot-spot-driven framework development process, which encompasses such a hot spot identification activity. Section 3.1 presents hot spot cards, which have proved to be a simple yet effective means for documenting and communicating hot spots. Below we outline the core activities in the framework development process and their relationships.

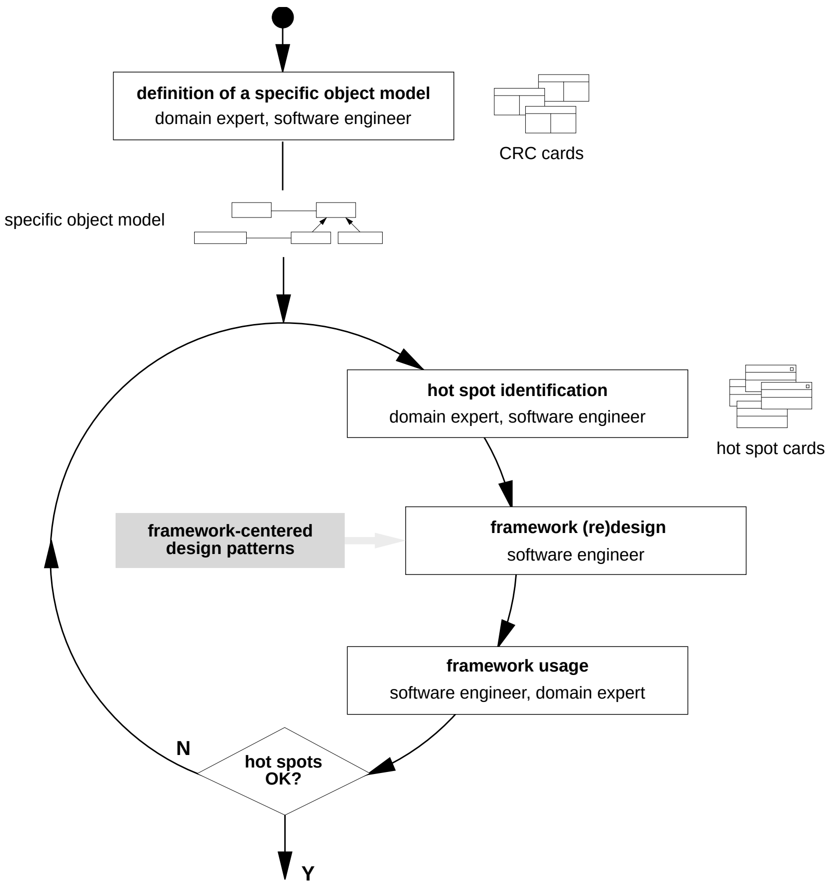

definition of a specific object model
domain expert, software engineer
CRC cards
specific object model
hot spot identification
domain expert, software engineer
hot spot cards
framework-centered design patterns
framework (re)design
software engineer
framework usage
software engineer, domain expert
N
hot spots OK?
Y

Figure 6 Hot-spot-driven development process (adapted from Pree, 1995).

Definition of a specific object model

State-of-the-art object-oriented analysis and design (OOAD) methodologies support the initial identification of objects/classes and thus a modularization of the overall software system. For example, the Unified Method (Booch, Rumbaugh, and Jacobson, 1997) picks out the best of Booch (1994), Rumbaugh (1991), and Jacobson (Jacobson, Ericsson, and Jacobson, 1996; Jacobson, Griss, and Jonsson, 1997). Class-Responsibility-Collaboration (CRC) cards (Beck and Cunningham, 1989) help in the initial identification of objects and their associations. Wilkinson (1996) discusses the usage of CRC cards in detail. Overall, object modeling is as challenging as any software development. Methodologies can only provide guidelines.

Object modeling requires primarily domain-specific knowledge. Software engineers assist domain experts in this activity. The distinction between domain expert and software engineer is a rather hypothetical one. It should just express the kind of knowledge needed. Of course, software engineers also have more or less deep domain knowledge.

Modeling a specific solution is already a complex and iterative activity where object models have to be refined until they meet the domain-specific requirements. This comprises object/class identification and probably the complete development of a specific software system.

Before starting the framework development cycle, it would, of course, help considerably to have already two or more object models of similar applications at hand: identifying commonalities would be a lot easier. Unfortunately, this is typically not the case.

Note that the actual framework development process must build on top of a specific yet mature object model.

Hot spot identification

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 in Section 2.3. 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. The synergy effect of essential construction principles paired with domain-specific hot spots is design patterns tailored to the particular domain (see Section 3.1).

The principal problem of this activity is that you cannot expect to get right answers if you do not ask the right questions. Often domain experts do not understand concepts such as classes, objects and inheritance, not to mention design patterns and frameworks. As a consequence, the communication between domain experts and software engineers has to be reduced to a common denominator. Hot spot cards (see Section 3.1) are such a communication vehicle, inspired by the few essential construction principles of frameworks outlined in Section 2.

A 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 less work than trying to find commonalities.

Framework (re)design

After domain experts have initially identified and documented the hot spots, software engineers have to modify the object model in order to gain the desired hot spot flexibility. Beginning with this activity, framework construction patterns as presented in the previous chapter assist the software engineer. In other words, patterns describing how to achieve more flexibility in a framework do not imply satisfactory frameworks, if software engineers do not know where flexibility is actually required. Hot spot identification meets the precondition to exploit the full potential of framework construction patterns.

Framework usage

A framework needs to be specialized several times, if not infinitely, in order to detect its weaknesses, that is, inappropriate or missing hot spots. The cycle in Figure 6 expresses the framework evolution process. Explicit hot spot identification by means of hot spot cards and framework construction patterns can contribute to a significant reduction of the number of iteration cycles.

3.1 Hot spot cards as means for capturing flexibility requirements

Domain experts can easily think in terms of *software functionality*, and they know how software functions can support, for example, various business processes. As a consequence, software engineers should try to obtain answers to the following questions from domain experts:

- Which aspects differ from application to application in this domain? A list of hot spots should be the result of this analysis.
- What is the desired degree of flexibility of these hot spots; that is, must the flexible behavior be changeable at run time and/or by end users?

As asking these questions directly fails in most cases, software engineers have to abstract from particular scenarios and look for commonalities and hot spot requirements. So software engineers will produce hot spot cards interactively with domain experts. Section 3.2 discusses some useful hints regarding hot spot identification.

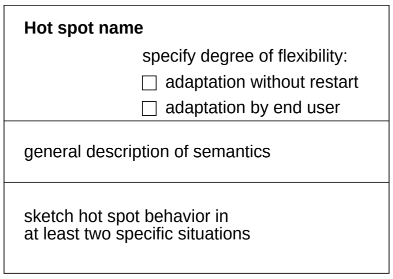
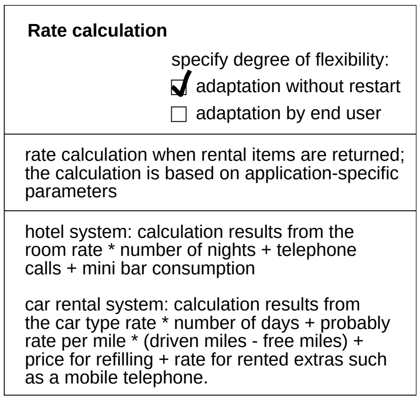

Hot spot name
specify degree of flexibility:
☐ adaptation without restart
☐ adaptation by end user
general description of semantics
sketch hot spot behavior in
at least two specific situations

Figure 7 Layout of a hot spot card.

A hot spot card first provides the hot spot name, a concise term describing the functionality that should be kept flexible, and specifies the desired degree of flexibility. Domain experts have to know that requesting run-time flexibility (adaptation without restarting) and/or the possibility of adaptation by the end user do not come for free. So this choice has to be made deliberately. Nevertheless, domain experts are tempted to demand maximum flexibility. Again, software engineers have to elaborate this aspect together with domain experts. The next section on the card summarizes the functionality. This section should abstract from details. Finally, the functionality has to be described in at least two specific situations so that software engineers can better grasp the differences.

For example, if a framework for rental software systems is to be developed that can easily be customized for hotels, car rental companies, and so on, a domain expert would identify the rate calculation as a typical function hot spot in this domain. Rate calculation in the realm of a hotel has to include the room rate, telephone calls and other extra services. In a car rental system different aspects are relevant for rate calculation. Figure 8 shows a corresponding hot spot card.



Rate calculation

specify degree of flexibility:

[✓ adaptation without restart]

☐ adaptation by end user

rate calculation when rental items are returned; the calculation is based on application-specific parameters

hotel system: calculation results from the room rate * number of nights + telephone calls + mini bar consumption

car rental system: calculation results from the car type rate * number of days + probably rate per mile * (driven miles - free miles) + price for refilling + rate for rented extras such as a mobile telephone.

Figure 8 Sample hot spot card.

Hot spot cards and essential construction patterns

Based on flexibility requirements specified as a stack of hot spot cards, software engineers have to transform the object model. In this step, framework-centered construction patterns, as presented in Section 2, assist the software engineer. Below we discuss the relationship between the information captured on hot spot cards and framework construction patterns.

Function hot spots closely correspond to hook methods and hook classes. A hot spot card contains information about the hot spot semantics and the desired degree of flexibility, but not to which class/subsystem the hot spot belongs. Nevertheless, the integration of function hot spots into the object model turns out to be quite straightforward. The precondition of a smooth integration is the appropriate granularity of a function hot spot. Function hot spots should correspond to methods or responsibilities. For example, a useless hot spot description expresses that database access should be flexible regarding a specific database.

In essence, a function hot spot with the right granularity implies that a hook method or a group of hook methods has to be added, either unified with the class where its template method resides or separated from it. Recall that a separation of template and hook classes forms the precondition of run-time adaptations. Table 1 summarizes how to transform the object model according to the flexibility information on a hot spot card.

Table 1 Transformation rules for a hot spot card.

adaptation ...	*adaptation by end user*	*object model transformation*
with restart	no	additional hook method
without restart	no	additional hook method in separate hook class
with restart	yes	additional hook method + configuration tool
without restart	yes	additional hook method in separate hook class + configuration tool

If none of the two check boxes (adaptation without restart, adaptation by end user) is marked, an extra hook method has to be introduced in an appropriate class. The hot spot semantics usually suffices for finding the class and its template method for integrating the hook. Let us take the rate calculation hot spot card as an example (see Figure 8), but assuming that none of the boxes is checked. We also assume that an abstract class RentalItem was added to the object model. We transform the object model according to the hot spot card by simply adding a calcRate() method to RentalItem. The corresponding template methods, such as printInvoice() and printOffer(), are in the same class (see Figure 9).

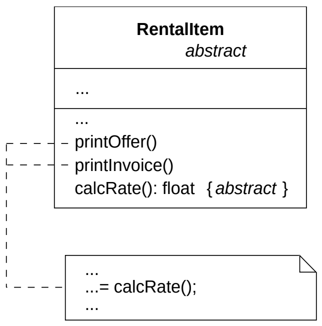


Figure 9 Hook method calcRate() resulting from a function hot spot.

If adaptations of the calculation engine should be possible without application restart, the additional hook method has to be put into a separate class. Figure 10 illustrates the object model transformation. Thus the rate calculation behavior of rental items can be changed by plugging in specific rate calculators.

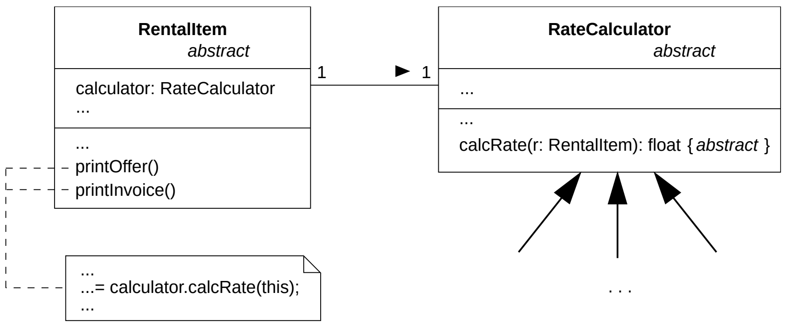

RentalItem
abstract
calculator: RateCalculator
...
...
printOffer()
printInvoice()
1
1
RateCalculator
abstract
...
...
calcRate(r: RentalItem): float {abstract}
...
...= calculator.calcRate(this);
...
. . .

Figure 10 Hook class RateCalculator resulting from a function hot spot.

How rate calculator objects are changed determines whether an end user should be able to accomplish this adaptation. One possibility would be to store the information regarding the calculation engine configuration in a resource file in plain text. Reservation systems built with the framework read this resource file on startup and when an end user requests this action explicitly. If the configuration file can only be edited by means of a text editor, many end users might refuse to effect such changes. On the other hand, end users could configure this system aspect if the resource file is edited interactively in a GUI editor.

Some hot spot cards might require no additional hook methods or classes at all. Resources and adequate editors might allow the achieving of the same flexibility as an object model transformation. For example, if only different prices influence the rate calculation algorithm, an elegant solution just stores this information in a resource file or database table.

Note that “adaptation by end user” does not necessarily mean that no programming effort is required. For example, if several rate calculators exist, chances are high that an end user finds an adequate one. Otherwise, a specific rate calculator would have to be implemented first.

Recursive template-hook combinations

So far, hot spot cards correspond to the unification/separation of template and hook methods. The alert reader will observe the lack of hot spot cards that reflect the recursive construction principles.

The design aspects covered by recursive template-hook combinations cannot be expressed in the reduced vocabulary of hot spot cards. This vocabulary deliberately focuses on functionality and excludes concepts such as class interface definition and inheritance. So it is up to the software engineer to apply recursive construction principles in order to produce a more elegant and flexible architecture.

Nevertheless, we can likely recognize many construction principles by analyzing the object model at hand. Relationships labeled *part-of*, *consists of*, *manages*, *owns* and the like indicate the GoF Composite pattern. If an abstract class becomes overloaded, software engineers could opt for behavior composition through object chains: Some of the behavior is put into separate classes so that this behavior can then be added by object composition.

Having the characteristics of object hierarchies and collaborating objects in mind, software engineers can intuitively detect situations where they can apply these recursive construction principles.

3.2 Hints for hot spot mining

The assumption is rather naive that you have perfect domain experts at hand, that is, those who produce numerous helpful hot spot cards just by handing out empty hot spot cards to them. In practice, most domain experts are absolutely not accustomed to answering questions regarding a generic solution. Below we outline ways to overcome this obstacle.

Examine maintenance

Most software systems do not break new ground. Many software producers even develop software exclusively in a particular domain. The cause for major development efforts that start from scratch comes from the current system, which has become hopelessly outdated. In most cases the current system is a legacy system, or, as Adele Goldberg (1995) expresses it, a *millstone*: you want to throw it away, but you cannot as you cannot live without.

As a consequence, companies try the development of a new system in parallel to coping with the legacy system. This offers the chance to learn from the legacy system. If you ask domain experts and/or the software maintenance crew where most of the effort was put into maintaining the old system, you'll get a lot of useful flexibility requirements. These aspects should become hot spots in the system under development. Often, a brief look at software projects where costs became outrageous in the past, is a good starting point for such a hot spot identification activity. Of course, those parts where flexibility is provided in an adequate way have to be transferred from the old system to the new one.

Investigate scenarios/use cases

Use cases (Jacobson et al., 1995, 1997), also called scenarios, turned out to be an excellent communication vehicle between domain experts and software engineers in the realm of object-oriented software development. They can also become a source of hot spots: take the functions incorporated in use cases one by one and ask domain experts about the flexibility requirements. If you have numerous use cases, you'll probably detect commonalities. Describe the differences between these use cases in terms of hot spots.

Ask the right people

This last advice might sound too trivial. Nevertheless, try the following: judge people regarding their abstraction capabilities. Many people get lost in a sea of details. Only a few are gifted to see the big picture and abstract from irrelevant details. This capability emerges in many real-life situations. Just watch and pick out these people. Such abstraction-oriented people can help enormously in hot spot identification and thus in the process of defining generic software architectures. So take at least some developers of the current system who have abstraction capabilities on board of the team that develops the new system.

4 Outlook

Above we discussed technical aspects of framework development by presenting the essential framework design patterns and the resulting hot-spot-driven development process. 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 intensively before jumping into framework development.

5 References

Beck K. and Cunningham W. (1989). A laboratory for object-oriented thinking. In *Proceedings of OOPSLA'89*, New Orleans, Louisiana

Booch G. (1994). *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummings

Booch G., Rumbaugh J. and Jacobson I. (1997) *Unified Method*. Documentation Set, Santa Clara, CA: Rational Software Corporation.

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

Goldberg A. (1995). *What Should We Learn? What Should We Teach?* Keynote speech at OOPSLA'95 (Austin, Texas); video tape by University Video Communications (http://www.uvc.com), Stanford, California

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

Jacobson I., Griss M. and Jonsson P. (1997). *Software Reuse: Architecture, Process, and Organization for Business Success*. Wokingham: Addison-Wesley/ACM Press

Jacobson I., Ericsson M. and Jacobson A. (1995). *The Object Advantage*. Wokingham: Addison-Wesley/ACM Press

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

Parnas D.L. (1976). *On the Design and Development of Program Families*. IEEE Transactions on Software Engineering, March 1976

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

Pree W. (1997). *Komponentenbasierte Softwareentwicklung mit Frameworks*. Heidelberg: dpunkt

Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. (1991). *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice-Hall

Wilkinson N. (1996). *Using CRC Cards—An Informal Approach to Object-Oriented Development*. Englewood Cliffs, NJ: Prentice Hall

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