Framework Component Systems: Concepts, Design Heuristics, and Perspectives

Wolfgang Pree, Gustav Pomberger

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, pomberger}@swe.uni-linz.ac.at

Franz Kapsner

Siemens AG, Corporate Research Division
Otto-Hahn-Ring 6, D-81739 Munich, FRG
Voice: +49 89-636-53364; Fax: +49 89-636-40 898
E-mail: franz.kapsner@zfe.siemens.de

Abstract. An appropriate combination of object-oriented programming concepts allows not only the development of single reusable components but also of semifinished architectures (= frameworks). The paper first discusses the concepts underlying frameworks and goes on to present design heuristics for developing such component architectures: Though design patterns are currently heralded as state-of-the-art in supporting framework development, a link between design patterns that capture and communicate proven object-oriented design and the framework development process is still missing. As pragmatic solution to this problem we introduce socalled *hot spot cards* to bridge the mentioned gap. These hot spot cards proved to be a valuable communication vehicle between domain experts and software engineers and thus helped to reduce the number of framework design iterations. An outlook sketches the perspectives and implications of framework technology.

Keywords. frameworks, software architectures, component-based software development, object-oriented programming, object-oriented design, design patterns, software reuse

1 Framework concepts

Object-oriented programming languages are used in many software projects in a manner similar to module-oriented languages with classes as a means for implementing abstract data types. Inheritance helps to adapt and thus reuse building blocks that do not exactly match the requirements. So adaptations are possible without having to change the source code.

In order to develop the full potential of object-oriented software construction in the realm of constructing reusable architectures, an *appropriate combination* of the

basic object-oriented concepts (i.e., object/class definition, inheritance in connection with polymorphism and dynamic binding) is necessary.

The key idea behind this approach is to find good abstractions of concrete classes. *Abstract classes* as discussed below represent such abstractions. They form the basis of frameworks, which are reusable application skeletons. Frameworks represent the highest level of reusability known today, made possible by object-oriented concepts: Not only source code but architecture design—which we consider as the most important characteristic of frameworks—is reused in applications built on top of a framework. Overall, frameworks enable a degree of software reusability that can significantly improve software quality.

Abstract classes. The general idea behind abstract classes is clear and straightforward:

- Properties (that is, instance variables and methods) of similar classes are defined in a common superclass.
- Classes that define common behavior usually do not represent instantiable classes but abstractions of them. This is why they are called *abstract classes*.
- Some methods of the resulting abstract class can be implemented, while only dummy or preliminary implementations can be provided for others. Though some methods cannot be implemented, their names and parameters are specified since descendants cannot change the method interface. So an abstract class creates a *standard class interface* for all descendants. Instances of all descendants of an abstract class will understand at least all messages that are defined in the abstract class.

  Sometimes the term *contract* is used for this standardization property: instances of descendants of a class A support the same contract as supported by instances of A.
- It does not make sense to generate instances of abstract classes since some methods have empty/dummy implementations.

The implication of abstract classes is that other software components based on them can be implemented. These components rely on the contract supported by the abstract classes. In the implementation of these components, reference variables that have the static type of the abstract classes they rely on are used. Nevertheless, such components work with instances of descendants of the abstract classes by means of polymorphism. Due to dynamic binding, such instances can bring in their own specific behavior.

The key problem is to find useful abstractions so that software components can be implemented without knowing the specific details of concrete objects.

Frameworks. Abstract classes form the basis of a framework. If abstract classes factor out enough common behavior, other components, that is, concrete classes or other abstract classes, can be implemented based on the contracts offered by the abstract classes. A set of such abstract and concrete classes is called a *framework*.

The term *application framework* is used if this set of abstract and concrete classes comprises a generic software system for an application domain. Applications based on such an application framework are built by customizing its abstract and concrete classes. In general, a given framework anticipates much of a software system's design. This design is reused by all software systems built with the framework.

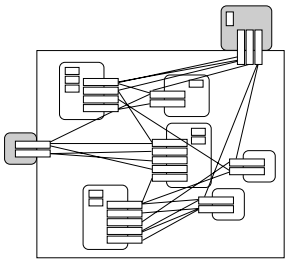

Fig. 1. Framework with flexible hot spots

In other words, 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*. Figure 1 illustrates this property of frameworks with the flexible hot spots in gray color. The overall framework comprises the standardized, i.e., frozen, domain aspects intertwined with the hot spots.

A framework-centered software development process comprises the creation and reuse of frameworks. This paper presents what we call *hot-spot-driven approach* as means to support framework development. The hot-spot-driven approach guides developers in the process of systematically incorporating experience captured and expressed in design patterns [6, 11, 12].

2 Hot-spot-driven framework design

We've experienced that successful framework development requires the explicit identification of domain-specific hot spots. The various aspects of a framework that cannot be anticipated for all adaptations have to be implemented in a generic way. As a consequence, domain experts have to be asked:

- Where is flexibility required? Which aspects differ from application to application in this domain? A list of hot spots should result from this analysis.
- What is the desired degree of flexibility of these hot spots, i.e., must the flexible behavior be changeable at run time?

Figure 2 schematically depicts what we call hot-spot-driven approach. State-of-the-art OOAD methodologies, such as those proposed by Booch [3], Coad and Yourdon [4], Jacobson [8], and Wirfs-Brock *et al.* [15], support the initial identification of objects/classes and thus a modularization of the overall software system. This initial step primarily requires domain specific knowledge. Software engineers assist in this activity. Of course, this first step is already an iterative one where object models have to be refined until they meet the domain-specific requirements.

2.1 Hot spot identification

Current OOAD methodologies neglect the importance of identifying hot spots as a basis for framework development. The schematically depicted framework development

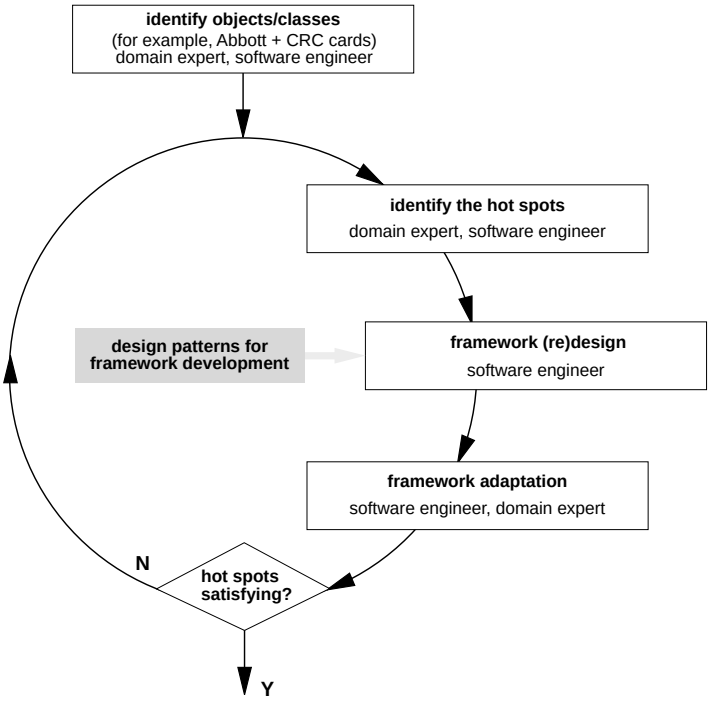

Fig. 2 Hot-spot-driven approach

process in Figure 2 starts with the identification of hot spots. Again, the knowledge of domain experts is considered to be the most valuable source in order to successfully complete this step. Software engineers might guide domain experts so that they describe hot spots in an adequate manner. As stated above, typical questions domain experts should be asked are: Which aspects differ from application to application? What is the desired degree of flexibility? Must the flexible behavior exist at run time? The software engineer can generate from the perspective of his current design more specific questions using the following rules:

- Having identified a class, find out from the domain expert how and whether objects would differ from application to application (whether there are subclasses).
- For a collection of type T ask whether it would be possible to have elements of various kinds in it. These may be subtypes of T.
- For a method defined for type T, find out for each subtype the dependencies of the method.

Domain experts might either identify *functions or data* as hot spots. For example, if a framework for rental software systems should be developed, that can easily be customized for hotels, car rental companies, etc., a domain expert would identify the rate calculation aspect as a typical hot spot in this domain. Rate calculation in the

realm of a hotel has to encounter the room rate, telephone calls, and other extra services. In a car rental system different aspects will be relevant for rate calculation.

The domain expert also has to assess the required flexibility of an identified hot spot, i.e., whether the hot spot behavior has to be adaptable at run time. For example, if a rental system framework should be targeted at rental companies with a world-wide operation run-time adaptability of the rate calculation is important.

The hot spot identification could be documented by what we call *hot spot cards*. Analogous to Class-Responsibility-Collaboration (CRC) cards [1], which help to define an object model, hot spot cards form the basis for transforming the object model into a domain-specific framework. Figure 3 depicts the proposed layout of a hot spot card. The hot spot name and the desired degree of flexibility is written on top of the card followed by a concise description of the hot spot semantics. In order to illustrate the hot spot behavior in application-specific situations at least two examples should be listed.

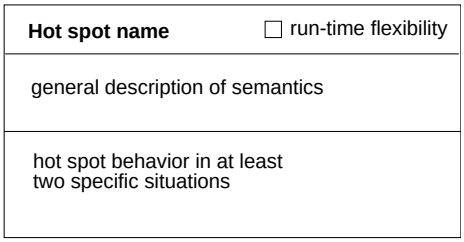

Fig. 3. Hot spot card

Figure 4 applies this scheme in order to illustrate how the description of a rate calculation hot spot could look like.

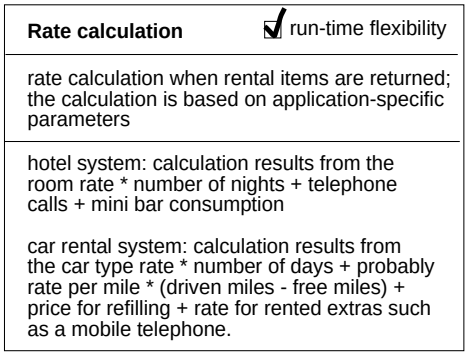

Fig. 4. Sample hot spot description of a function

An example of a data hot spot in the rental system framework would be the rental items. It is not necessary to specify whether a data hot spot should be flexible at run time because class definition is not possible at run time in static languages such as C++, Smalltalk, Oberon and Eiffel.

2.2 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. In this step design patterns for framework-centered software development as discussed in [6, 11, 12], assist the software engineer (see Figure 2). Below we describe how frameworks can be constructed in a systematic manner once the hot spots have been identified. Thus we consider hot spot identification as a precondition in order to exploit the potential of design pattern approaches.

Framework construction principles—metapatterns

Actually, constructing frameworks by combining the basic object-oriented concepts proves quite straightforward. We use the term *metapatterns* for a set of design patterns that describes how to construct frameworks independent of a specific domain. These framework construction principles together with numerous examples are described in detail in [11]. We pick out some fundamental metapatterns in order to illustrate their importance in the realm of hot-spot-driven framework development.

Template and hook methods. In general, template methods implement the frozen spots and hook methods implement the hot spots of an application framework. Note that the term template method, or simply template, must not be confused with the C++ template construct, which has a completely different meaning.

With some imprecision, we can say that complex methods called *template methods* can be implemented based on elementary methods which are called *hook methods*. Template methods are a means of defining abstract behavior or generic flow of control. The abstract behavior is adapted by overriding hook methods.

Template and hook classes. We call the class that contains the hook method(s) the *hook* class, and the class that contains the template method(s) a *template class*. In other words, a hook class *parameterizes* the corresponding template class.

There are only a few ways how to define a relationship between the objects of template and hook classes. Pree [11] describes these *composition metapatterns* in detail. The basic composition metapatterns are outlined below.

In the *Unification metapattern* template class and hook class are unified, that is, one class contains template and hook methods (see Figure 5(a)). An adaptation of the template method requires the definition of a subclass where the hook method is overridden. As subclasses cannot be defined at run time in most object-oriented languages, the Unification metapattern offers no run-time flexibility.

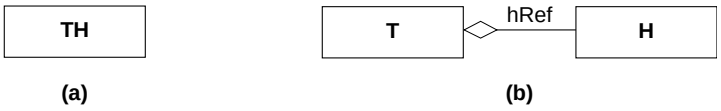

Fig. 5. (a) Unification and (b) 1:1(N) Connection metapatterns

In the *1:1(N) Connection metapattern* an object of a template class refers to exactly one object or a collection of objects of its hook class. No inheritance relationship between template class and hook class exists (see Figure 5(b)). If subclasses of `H` already exist, the behavior of a `T` object is changed by plugging in a specific `H` object. Such an assignment to the instance variable `hRef` of a `T` object can be accomplished at

run time. The 1:1(N) Connection metapattern corresponds to the Bridge pattern [6] and is, for example, applied in the State pattern, Command pattern, and Observer pattern [6].

From hot spots to framework design. View the recommended hot spot identification in connection with essential construction patterns from the following perspective: Framework quality strongly correlates with adequate architectural flexibility. Thus an explicit capturing of flexibility requirements together with a quite systematic transformation of a specific object model indeed proved to contribute to a more systematic framework development process.

There are essentially two motivations to make hot spot identification an explicit activity in the development process: One is that design patterns, presented in a catalog-like form such as in Gamma *et al.*, 1995, mix construction principles and domain specific semantics. 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 requirement analysis provides this information leading to a synergy effect of essential construction principles paired with domain-specific hot spots. The result are design patterns tailored to the particular domain.

Another reason, why explicit hot spot identification helps, can be derived from the following observations of influencing factors in real-world framework development: Usually, there are not two or more similar systems around 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 siginficantly 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. Hot spot cards represent a first vehicle that is based on the considerations stated above. In the following we exemplify how to transform the flexibility requirements captured in hot spot cards.

Data hot spots require the introduction of an abstract class. In the rental system framework the abstract class RentalItem is defined (see Figure 6).

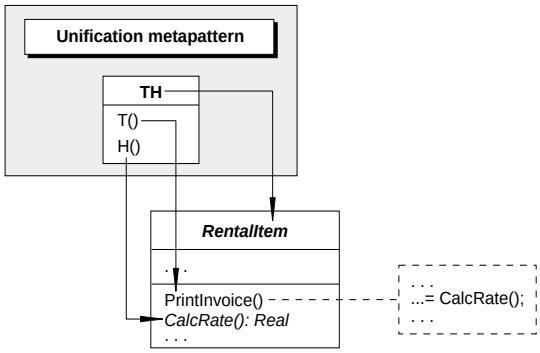

Fig. 6. A hot spot based on the Unification metapattern

Hook methods closely correspond to hot spot functionality. For example, if how rental rates are calculated for a rental item has to be kept flexible, the Unification metapattern could be chosen in the framework design as shown in Figure 6. In this design the rate calculation cannot be kept flexible at run time.

In order to allow run-time adaptations the 1:1 Connection metapattern would be appropriate as shown in Figure 7.

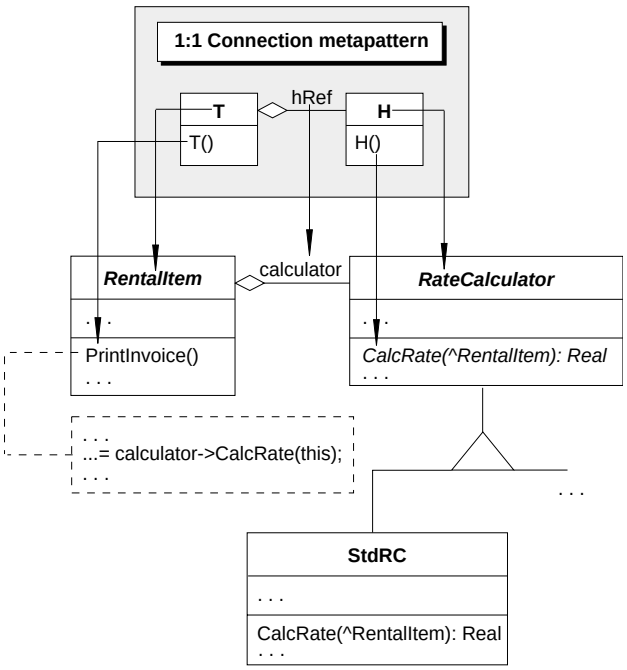

Fig. 7. A hot spot based on the 1:1 Connection metapattern

Note that hot spots with run-time flexibility typically imply an additional abstract class that corresponds to the hot spot semantics. Thus additional flexibility implies extra costs as the design artifact turns out to be more complex. In the example shown in Figure 7 the abstract class RateCalculator resulted from the corresponding hot spot description shown in Figure 4: run-time changes should be possible. The behavior of template method PrintInvoice() depends on the associated RateCalculator object. Any specific rate calculator attatched to a rental item at run time can change the behavior of template method PrintInvoice() in a RentalItem object.

Note that changing the behavior of a software system at run-time by defining completely new classes requires further support by the language/run-time system and the operating system. For example, in order to define a new subclass of RateCalculator and associate an instance of that class to rental items, the operating system has to support dynamic linking. Furthermore, meta information about objects and classes is required to instantiate newly linked classes.

2.3 Framework adaptation and evolution

A framework needs to be specialized several times in order to detect its weaknesses, that is, inappropriate or missing hot spots. The framework evolution process is described in [14]: “Good frameworks are usually the result of many design iterations and a lot of hard work. Designing a framework is like developing a theory. The theory is tested by trying to reuse a framework.”. The cycle in Figure 2 expresses the framework evolution process. Explicit hot spot identification by means of hot spot cards together with design patterns help to reduce the number of iteration cycles.

3 Hints for hot spot mining

Of course, the assumption is rather idealistic that you have perfect domain experts at hand, that is, those who produce a punch of helpful hot spot cards just by handing out empty hot spot cards to them.

In practice, most domain experts are absolutely not used to answering questions regarding a generic solution. The current project culture forces them to do requirements analysis and system specifications that match exactly one system. Vague or generic statements are not welcome. Below we outline ways to overcome this obstacle.

Take a look at maintainance. Most of the 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 [7] expresses it, a *millstone*: you want to put 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 maintainance problems of the legacy system. If you ask domain experts and/or the software engineering 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 outrageuos in the past, is a good starting point for such a hot spot mining activity.

Investigate scenarios/use cases. Use cases [8, 9], 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 regarding the flexibility requirements. If you have numerous use cases, you’ll detect probably 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 shows off in many real-life situations. Just watch and pick out these people. Such abstraction-oriented people can help enormously in hot spot mining and thus in the process of defining generic software architectures.

4 Outlook

Though object/framework technology is touted as yet another one and only path to true knowledge, it simply is not. Object/framework technology tries to unify the lessons learned in programming methodology over the past 30 years. Nevertheless, too many problems are still encountered. For example, if the interface of (abstract) framework classes is changed applications built on top of the framework are rippled. This is commonly known as *fragile base class problem.*

Recent definitions of component standards (CORBA, COM-OLE, (D)SOM-OpenDoc) can be viewed as an attempt to cope with this problem. But these standards are so controversial that even the members of OMG (Object Management Group, a group of object technology vendors) disagree on the definition of the fundamentals of object technology. Lewis *et al.* [10] state that "the OMG has found some cracks in the foundation of object technology."

Furthermore, tools for better understanding frameworks and for testing components added to a framework would be crucial. Currently such tools are not adequate or even not existing.

Nevertheless, we think that these obstacles can be removed and that framework technology will be the *enabling technology that underlies future software systems*. In essence, component-based software development [2] might have the meaning then that companies develop components that specialize frameworks. As frameworks are well suited for any domain where numerous similar applications are built from scratch again and again, they will exist for numerous domains ranging from commercial and technical software systems to advanced applications such as intelligent agents. In many cases, frameworks will be adapted by just plugging in various components so that end users are able to do configuration jobs.

The only thing that will probably change is the meaning of the term framework. Lewis *et al.* [10] compare such a term evolution with what happened to the term *compiler*: "In 1954, a compiler was a program that collected subroutines together to construct another program. Today, a compiler is a program that translates a high-level language into machine language. The term has taken on entirely different meaning than that intended by its author. The same fate awaits the term framework." [10]

For sure, framework development requires a radical departure from today's project culture. Framework development does not result in a short-term profit. On the contrary, frameworks represent a long-term investment. The proposed hot-spot-driven approach is aimed at exploiting the potential of design patterns for framework development. Experience has proven that the explicit identification of hot spots together with a systematic transformation of the corresponding domain object model contribute to finding appropriate methods and abstractions faster so that the number of redesign iteration cycles is reduced.

References

1. Beck K. and Cunningham W. (1989). A laboratory for object-oriented thinking. In Proceedings of *OOPSLA'89*, New Orleans, Louisiana
2. Berg K., Franz M., Hollunder B., Marek B., Pree W. (1995) *Component-Based Software Development—A State-of-the-art Survey*. Siemens AG Munich; technical report

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

4. Coad P. and Yourdon E. (1990). *Object-Oriented Analysis.* Englewood Cliffs, NJ: Yourdon Press

5. Coplien J. (1992). *Advanced C++: Programming Styles and Idioms*. Reading, Massachusetts: Addison-Wesley

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

7. 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

8. Jacobson I., Christerson M., Jonsson P. and Overgaard G. (1993). *Object-Oriented Software Engineering.* Wokingham: Addison-Wesley/ACM Press

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

10. Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J., Schmucker K. (1995) *Object-Oriented Application Frameworks.* Manning Publications, Prentice Hall

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

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

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

14. Wirfs-Brock R.J. and Johnson R.E. (1990). Surveying current research in object-oriented design. *Communications of the ACM*, **33**(9)

15. Wirfs-Brock R., Wilkerson B. and Wiener L. (1990). *Designing Object-Oriented* Software. Englewood Cliffs, NJ: Prentice-Hall