Systematic Framework Design by Generalization

How to deduce a hot spot implementation from its specification.

A framework [5] is a generic application that allows the creation of different applications from an application (sub)domain. Due to the inherent flexibility and variability of a framework, framework design is much more complex than application design [4]. Our experience (first gained when designing a manufacturing framework [8, 9]) shows that the complexity of framework design is reduced by separating clearly different issues: the design of a class model for an application from the framework domain; the analysis and specification of the domain variability and flexibility; and its stepwise implementation by a sequence of generalizing transformations. Since application design is a well-known activity, we will concentrate on the specification of the variable aspects, on the design of a (local) class structure that provides each with the required variability, and on how to transform a (global) class structure for generalization. For more details see [10].

Hans Albrecht Schmid

A variable aspect of an application domain is called a hot spot [6, 8]. Different applications from a domain differ from one another with regard to some (at least one) of the hot spots. An application supplies, for each hot spot, one (or several) of the different possible alternatives of the variability. All different alternatives to be supplied for a hot spot have a responsibility, named R in Figure 1, in common.

A hot spot lets you "plug-in" an application-specific class or subsystem, either by selection from a set of those supplied with a black-box framework, or by programming a class or subsystem in a white-box framework (see Figure 1 left). In this way, you create an application from the framework.

The variability required from a hot spot is classified by the following characteristics (from which the implementation will be derived directly):

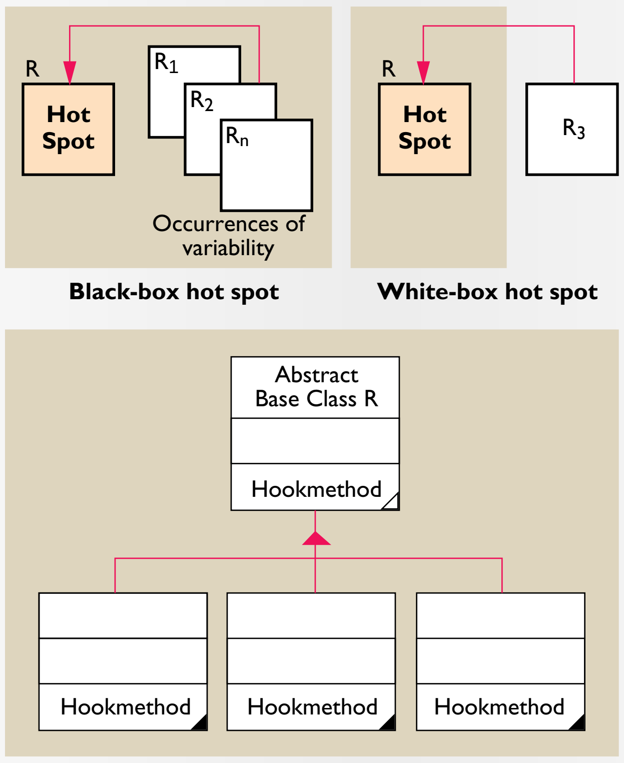

Figure 1. Hot spot in a black-box and white-box framework (left) and hot spot subsystem (right)

- The common responsibility R that generalizes the different alternatives.

- The different alternatives that realize R.

- The kind of variability required. You might require the following variability:
  Example 1: a common interface with different implementations, like a set collection class with each a linked-list, binary tree and hash implementation;
  Example 2: different kinds of bank accounts where the operation to withdraw money defines a common sequence of actions like check withdrawer identity, check amount, and take off amount. However, the check amount may differ for different kinds of accounts; and
  Example 3: a document with a tree structure of variable depth and size, as chapters, sections and subsections of text elements, that provides uniform services like "get next text element."

- The multiplicity gives the number (either one, or n) and structuring (for n alternatives: either chain-structured or tree-structured) of the alternatives that may be bound to a hot spot. The multiplicity characteristic is directly related to the kind of variability. For Example 1 and Example 2, the multiplicity is one, whereas for Example 3, it is n and tree-structured.

- The binding time characterizes the point of time at which an alternative is selected: at the time of creating an application by the application developer; at run time, by an end user. At run time, the binding may be done either once and fixed, or repeatedly.

Hot Spot Subsystem

A hot spot is implemented by a hot spot subsystem [8]. Note that we use the Coad/Yourdon notation [2] as generated from the tool ObjectiF to represent class diagrams. A hot spot subsystem contains (see Figure 1, right):

- A (typically abstract) base class, which defines the interface for common responsibilities.
- Concrete derived classes, each representing one of the different alternatives for the variable aspect.
- Possibly additional classes and relationships.

Also, a polymorphic reference typed with the base class is contained in or attached to a hot spot subsystem. Setting the reference to a subclass object, which is done when configuring the hot spot subsystem, lets you bind the hot spot. A method calling a base class operation (called template method and hook method [6]) via this reference is dynamically bound to the subclass method executed. The effects of calling a hook operation are generic: they depend on the way the hot spot subsystem has been configured. Thus, a hot spot subsystem introduces variability that is (usually) transparent to the remainder of the framework.

We classify hot spot subsystems similarly to metapatterns [6], into the different categories: *non-recursive*, in which a requested service is provided essentially from only one object; *chain-structured (1:1) recursive*: wherein a requested service may be provided by a chain of subclass objects; and *tree-structured (1:n) recur-*

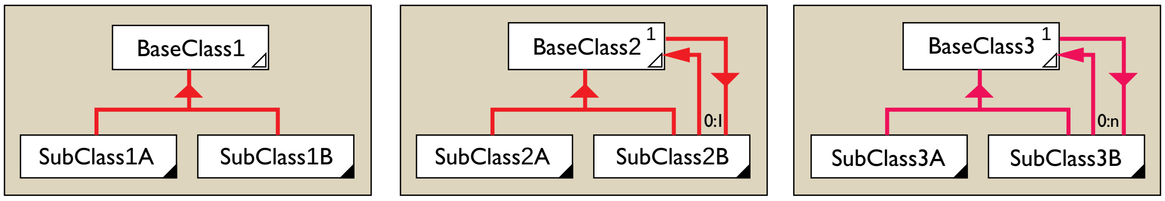

Figure 2. A non-recursive, a chain-structured recursive, and a tree-structured recursive hot spot subsystem

sive, meaning a requested service may be provided by a tree of subclass objects.

Recursive hot spot subsystems have a contains-relationship among a subclass (or the base class) and the base class (see Figure 2). Design patterns, which describe typical, common, and frequently observed relationships among classes, help determine the detail structure of a hot spot subsystem. Each of the design patterns from [4] (except for the Singleton, Facade, Flyweight and Memento patterns) provides a different kind of variability. Thus, a design pattern presents a proved solution for how to internally structure a hot spot subsystem in detail. All design patterns are non-recursive hot spot subsystems, except Chain of Responsibility and Decorator, which are chain-structured recursive, and Composite and Interpreter, which are tree-structured recursive hot spot subsystems.

Binding of hot spots at run time (by interaction with an end user or by a lookup in tables or in a plan [8, 9]) requires a supporting class structure. In contrast, binding them at the time of application creation does not require this since this is a kind of meta-activity from the application programmer side.

From Hot Spot Characteristics to Hot Spot Subsystem Structure

You may deduce the detailed class structure of a hot spot subsystem straightforwardly from the hot spot characteristics.

- The multiplicity directly indicates the hot spot subsystem category.
- Look for a design pattern in the respective hot spot subsystem category that provides the required kind of variability. You may need to add classes and relationships to the base class and subclasses, and refine the classes and collaborations following the design pattern. When an appropriate design pattern is not found, you design the detailed structure by refining the selected hot spot subsystem category. For example: The multiplicity and variability requirements of Example 1 are met by a non-recursive subsystem structured according to interface inheritance, those of Example 2 by a non-recursive subsystem structured fol-

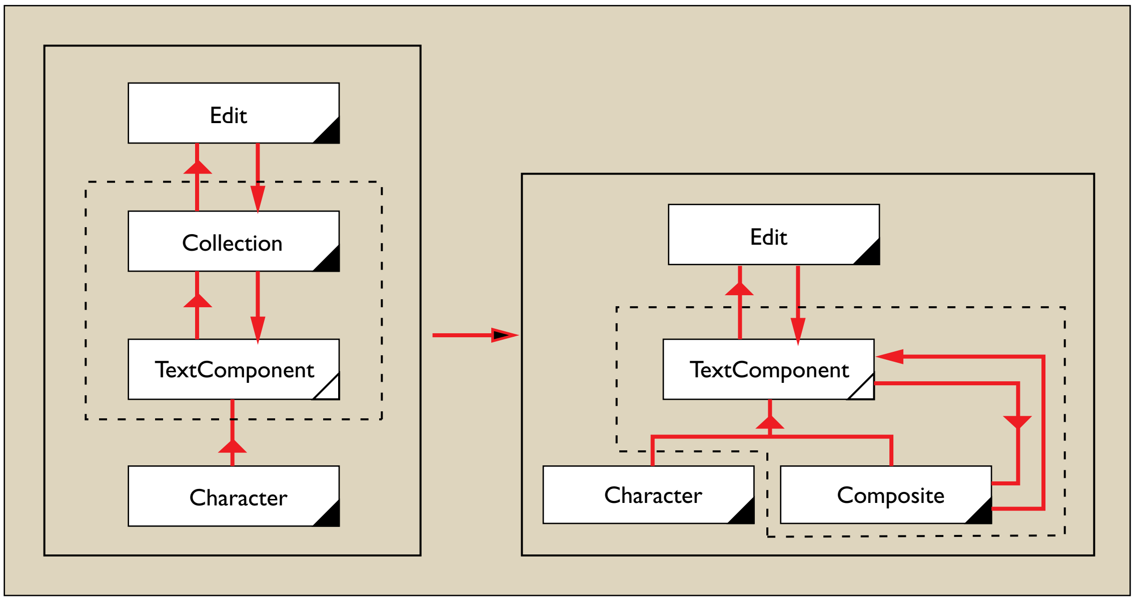

Figure 3. Hot spot transformation generalizing the aggregation of TextComponent

lowing the template method design pattern, and those of Example 3 by a tree-structured recursive subsystem structured following the composite design pattern.
- The abstract base class interface realizes the common responsibilities R and the pattern-related responsibility. A derived class implements one of the different alternatives of the variable aspect and the pattern-related responsibility.
- The binding time indicates if a class structure in support of binding is to be added or not.

Generalization Transformation

The variability of a hot spot is introduced into a framework by generalizing the class structure [8] (compare generative design patterns [1]). The class structure contains, before generalization, a specialized class (or classes) that has a direct and fixed relationship to other classes, representing the respective aspect as a frozen spot. When a generalization transformation is performed, this specialized class (possibly also directly related classes) is replaced by a hot spot subsystem, and the direct relationship between a client and a specialized class is replaced by a polymorphic (indirect) relationship.

Figure 3 takes up Example 3. It shows how the fixed relationship among an Editor and a collection of TextComponents (left side) is generalized to a variable relationship among an Editor and a variable aggregation hierarchy of TextComponents (right side). This is done by replacing the specialized class Collection and TextComponent by the hot spot subsystem derived for Example 3.

Conclusion

Decomposing the complex tasks of framework design into the following set of basic activities makes framework design easier:

1. The modeling activity designs the class structure of a (specialized, fixed) application from the framework domain.
2. The hot spot analysis activity collects the hot spots and describes the characteristics of each hot spot in a "hot spot variability requirements" specification.
3. The generalization activity generalizes a specialized class structure to incorporate the domain variability, by applying by a sequence of generalization transformation steps, each one for a hot spot. In each step, a specialized class is replaced by a hot spot subsystem, the structure of which follows from the hot spot characteristics.

Outlook

When developing a framework, don't plan to do all design activities described here in one development cycle. Framework development should be based on experience, nobody will develop a useful framework from scratch in one development cycle [7]. Therefore, the design activities should be distributed over different development cycles. Very often an application class structure will be designed and (parts of it) implemented in the initial development cycle(s). Then a rough analysis of all hot spots will be performed, followed by a hot spot detail analysis, class structure generalization and implementation, each for one or a few hot spots at a time, in subsequent development cycles. Though not part of the ideal development, in practice restructuring cycles [3] may be required if the requirements were not well understood or if preplanning was not done or did not work out properly for a hot spot. Systematic framework design, however, will reduce the need for restructuring. C

References

1. Beck, K. and Johnson, R. Pattern generate architectures. In *Proceedings of ECOOP 1994, Springer Lecture Notes in Computer Science*, Berlin. 1994.
2. Coad, P. and Yourdon, E. *Object Oriented Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
3. Foote, B. and Opdyke, W. Life cycle and refactoring patterns that support evolution and reuse. In *Pattern Languages of Program Design*, Addison-Wesley, Reading, Mass., 1995.
4. Gamma, E., Helm, E., Johnson, R.R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.
5. Johnson, R.E. and Foote, B. Designing reusable classes. *J. Object-Oriented Programming 1*, 2 (June 1988), 22–35.
6. Pree, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Reading, Mass., 1994.
7. Roberts, D. and Johnson, R. Evolve frameworks into domain-specific languages. In *Pattern Languages of Program Design 3*, Addison-Wesley, Reading, Mass., 1997.
8. Schmid, H.A. Design patterns for constructing the hot spots of a manufacturing framework. *J. Object-Oriented Programming 9*, 3 (June 1996), 25–37.
9. Schmid, H.A. Creating the architecture of a manufacturing framework by design patterns. In *Proceedings of OOPSLA'95*, ACM, NY 1995, pp. 370–384.
10. Schmid, H.A. Framework design by systematic generalization: From hot spot specification to hot spot subsystem. In *Implemenatation in Object-Oriented Application Frameworks*. M. Fayad, D.C. Schmidt, R. Johnson, Eds., Wiley, NY, to appear.

Hans Albrecht Schmid (schmidha@fh-konstanz.de) is a professor of computer science at the Fachhochschule Konstanz in Konstanz, Germany.

The Deutsche Forschungsgemeinschaft (DFG) provided partial support for this project.