Managing Object-Oriented Frameworks with Specialization Templates

Markku Hakala[1], Juha Hautamäki[1], Jyrki Tuomi[1], Antti Viljamaa[2], Jukka Viljamaa[2], Kai Koskimies[3], Jukka Paakki[2]

Abstract. The concept of a specialization template is introduced. It is shown that specialization templates are useful in the design and documentation of object-oriented frameworks. A prototype tool for Java frameworks is implemented based on a formalization of a specialization template. The tool helps a framework developer to manage the hot spots and the specialization rules of the framework, and an application developer to specialize the framework. The approach has been applied to a realistic case study, a Java framework for developing graphical editors.

1 Introduction

Within the object-oriented paradigm, *framework* has become a popular catchword for a product-line architecture. Nonetheless, frameworks are generally not well understood as software products. The term framework is used for a wide spectrum of architectures without always recognizing the features that actually characterize a framework. The design process of a framework is currently not mastered by software professionals, a fact harshly revealed by the long, unpredictable and iterative development projects of frameworks. The use of a framework for producing an application is often problematic due to the mere size and complexity of the framework, making it hard to understand exactly which parts must be specialized and how. Tool support for making the development and use of frameworks easier is just taking its first steps ([FMW97], [MDE97]). Taking into account the lack of understanding of the framework concept itself, it is hardly surprising that the tool support is inadequate so far.

There are both impressive success stories and (less published) crushing catastrophes in making use of frameworks in the industry. For mainly positive experiences, see e.g. [Cas95], [SBF96], [Joh92], [Lew95] and [CACM97]. We feel that frameworks have the potential for truly powerful software reuse, but as everything incompletely understood they also contain a significant risk. In this paper we will propose a new concept, *specialization template*, that helps to understand a framework and its design process. We also show that this concept can be used as the basis of tool support for frameworks.

[1] University of Tampere, Box 607, 33101 Tampere, Finland (email {mh, csjuha, jjt}@cs.uta.fi)

[2] University of Helsinki, Box 26, 00014 University of Helsinki, Finland (email {antti.viljamaa, jukka.viljamaa, jukka.paakki}@cs.helsinki.fi)

[3] Tampere University of Technology, Box 553, FIN-33101 Tampere, Finland (email: kk@cs.tut.fi)

In the next section we will introduce the concepts of design contract and specialization template as basic design artifacts and documentation means for a framework. In section 3 we discuss the role of specialization templates in the design of a framework. The basic design of a practical tool that builds on the idea of a specialization template is presented in section 4. Finally, we summarize the benefits of the approach in section 5.

The tool presented here is FRED (Framework Editor), a recent result of research carried out at the universities of Helsinki and Tampere. An early, significantly different version of FRED was reported in [Hak98]. As a realistic case study, FRED has been applied to JHotDraw, a Java framework for graphical editors [Gam98]. FRED and the JHotDraw application (along with a tutorial) are freely available at http://www.cs.helsinki.fi/research/fred.

2 Design contracts and specialization templates

Basically, a *design contract*[4] is a variant of the contract concept introduced by Holland [Hol93]. It is also closely related to a design pattern [Gam95]. A design contract is an application-independent description of a certain OO design aspect. In contrast to a traditional design pattern, it does not necessarily describe a solution to a frequently occurring problem (or to any "problem" for that matter). In a sense, a design contract is a design pattern released from its semantic burden. It describes simply a set of classes, together with certain attributes, methods and relations. Further, a design contract specifies the (structural) conditions that any realization of the contract must fulfil. Naturally any design pattern can be easily converted to a design contract, but not vice versa.

The idea of a design contract is close to Holland's notion of a contract. Since we want eventually to use design contracts as the basis of our tool, we must formalize them as a language, resembling Holland's Contract language. The differences stem from our aim to support the specialization process using the information in a design contract, rather than to specify general reusable program fragments. When compared to Holland's contracts, design contracts are less specific about the method bodies (algorithms). Further, we introduce static constraints that can be used as guidance in the specialization phase, rather than run-time invariants. All this means that we consider contracts as static, structural specifications rather than as semantically meaningful abstract program fragments.

Because of its character, the informal description of a design contract is more compact than that of a design pattern. The central parts of a design contract are:

---

[4] This concept is not directly related to "design-by-contract" as proposed by Meyer.

- *Name*: the name that identifies the design contract.
- *Base contract*: the possible design contract which this design contract is derived from.
- *Structure*: the static structure of the participants of the contract, in terms of a UML class diagram. The names appearing in the structure are *roles*, i.e. placeholders that can be bound to actual code items.
- *Constraints*: the constraints that must be followed in any realization of the structure, when the roles of the structure are bound to application specific items.

Often design contracts can be organized into hierarchies in which a more specific contract extends a more general contract, adding certain classes, attributes, methods (roles) or constraints. For example, there might be a simple design contract consisting of two classes with inheritance relationship; this contract could then be extended by another contract with a structure more specific to it. The possible design contract serving as the base of extension is given in the "base contract" part. This is equivalent to Holland's notion of "refinement".

We describe design contracts formally using a textual language called Cola (Contract language). To get some idea of the flavor of Cola, consider the following (somewhat simplified) Cola specification of the Composite design contract:

```
contract CompositeContract{
  single type Component { method operation ; }
  type Leaf inherits <Component C> {
    named "Leaf" + C ;
    single method operation overrides <C~operation O> ;
  }
  single class Composite inherits <Component C> {
    named "Composite" + C ;
    single method operation overrides <C~operation O> {
    }
    single method add {
      takes C child ;
    }
    single method remove {
      takes C child ;
    }
    single method get {
      returns Component ;
      takes int i ;
    }
    single field components {
      returns java.util.Vector ;
    }
  }
}
```

Since Java is the underlying implementation language in FRED, Cola deliberately follows Java's syntactic style. The pattern includes three top-level roles: `Component`, `Leaf`, and `Composite`. A main constraint of a role is its *cardinality*. Cardinality denotes the number of Java entities that must or can be bound to the role. The modifier `single` states the role's cardinality: one suitable Java type (an interface or a class) must be bound to the role in every instance of `CompositeContract`. `Component` in turn includes a method role called `operation`. Its cardinality is relative to the parent role. The role has no cardinality modifiers, implying that there must be 1 – *n* Java methods in each Java type bound to the role `Component`.

For example, the role `Leaf` represents Java types whose objects cannot have children. It defines a *parameter C* of type `Component`. If a role has a parameter, its cardinality is relative to the number of Java entities bound to the role referred to by the parameter. In this case, there must be one Java entity bound to the role `Leaf` for every Java entity bound to the role `Component`. The parameter can also be used in the constraints associated with the role as well as in its subroles. The `inherits` constraint in `Leaf` specifies that each Java type playing the role of `Leaf` must inherit (either directly or indirectly) a Java type bound to the role referred to by the parameter *C*, i.e. the role `Component`. The `named` constraint says that these leaves must have names beginning with the string constant “Leaf” followed by the name of the Java type bound to the parameter.

Design contracts are an essential part of *specialization templates*. A specialization template is a framework-specific design solution to a particular flexibility requirement. Using the terminology of Pree [Pre95], a specialization template specifies a *hot spot* in a framework. Logically a specialization template corresponds to Holland's concept of "conformance", although we emphasize different aspects. A specialization template is based on a design contract: essentially it binds a design contract to a particular framework and its flexibility requirements. The central parts of a specialization template are:

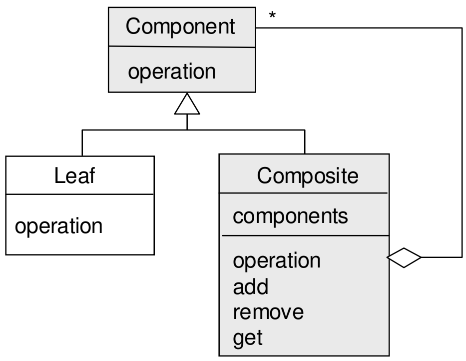
- *Name*: the name of the specialization template.
- *Flexibility requirement*: the flexibility requirement of the framework, in terms of application variance, that gives rise to this specialization template.
- *Design contract*: the design contract this specialization template is based on.
- *Structure*: the structure of the contract, presented so that the classes that are bound to actual (Java) classes are shaded.
- *Bindings*: the bindings of certain roles to actual Java items.
- *Constraints*: the constraints related to the unbound roles.
- *Specialization hints*: additional hints to guide the application developer in writing the specialization code (e.g. method bodies).
- *Example*: a representative example of a particular specialization.

Note that a design contract is framework-independent whereas a specialization template is framework-dependent. A specialization template binds a part of the contract structure to certain entities (classes, attributes, methods) in the Java framework. The more a specialization template binds, the less freedom is left to the application developer. On the other hand, if the application developer leaves certain roles unbound, the result is a new, less general specialization template, associated with a less general framework.

Indeed, we consider framework development as a continuum of specialization, terminating in an executable application (see next section).

As an example, we present a simplified specialization template associated with the JHotDraw framework. We omit the design contract (Composite) this template is based on: the contract is a fairly straightforward reformulation of the Composite design pattern [GHJV95].

- *Name*: CompositeFigure.
- *Flexibility requirement*: It must be possible to define and add new types of figures appearing as nodes in the graphs manipulated by the editor.
- *Design contract*: Composite.
- *Structure*:



- *Bindings*: Component → Figure, Composite → CompositeFigure, operation → draw, add → add, remove → remove, get → get, components → componentFigures. (Names on the right-hand sides refer to entities in the JHotDraw framework.)
- *Constraints*:
  - any number of Java classes can be bound to Leaf
  - any class bound to Leaf must override the method bound to operation
  - the method bound to operation must implement the drawing of the figure.
- *Specialization hints*: The figure can be defined also as an icon as follows: …
- *Example*: Typically, the drawing operation is given as follows: …

Specialization templates are not formally defined using a textual language, but they are instead constructed interactively using the FRED interface.

3 Specialization templates in framework design

The crucial part of the analysis phase in framework development is stepwise generalization ([KoM95], [Sch97]). The process begins with the specification of an example application in the intended domain of the framework. In each generalization step, some concepts in the requirements specification are generalized, transforming the application specification into a framework specification. When this process is continued, more and more general framework specifications emerge, until eventually a suitable level of abstraction is reached.

Each generalization of a concept gives rise to a flexibility requirement. Essentially, a flexibility requirement stores the information concerning a single concept generalization by specifying the concept that can have variations. Roughly, a generalization of concept *A* to concept *B* gives rise to flexibility requirement "It must be possible to define and add variants of *B*" and to example specialization *A*. Each generalization step in the analysis stage typically comprises several flexibility requirements.

In the design phase, the most abstract framework specification is first implemented ([KoM95], [Sch97]). Each flexibility requirement associated with the specification is transformed into a specialization template. Note that the flexibility requirement itself will be part of the template. An example specialization is also directly available from analysis. The framework developer then designs an architecture that fulfils the flexibility requirement, either using an existing design contract or creating a new one. Possible specialization hints and example specializations are written down. The framework classes involved in the template are implemented and bound to its roles.

After implementing in this way the most abstract framework, the next more concrete framework specification is implemented in the same way. In the ideal case the next framework can be obtained as a specialization of the previous one, exploiting its specialization templates. This results in a layered framework, with increasing level of concreteness. The design process is depicted in Figure 1.

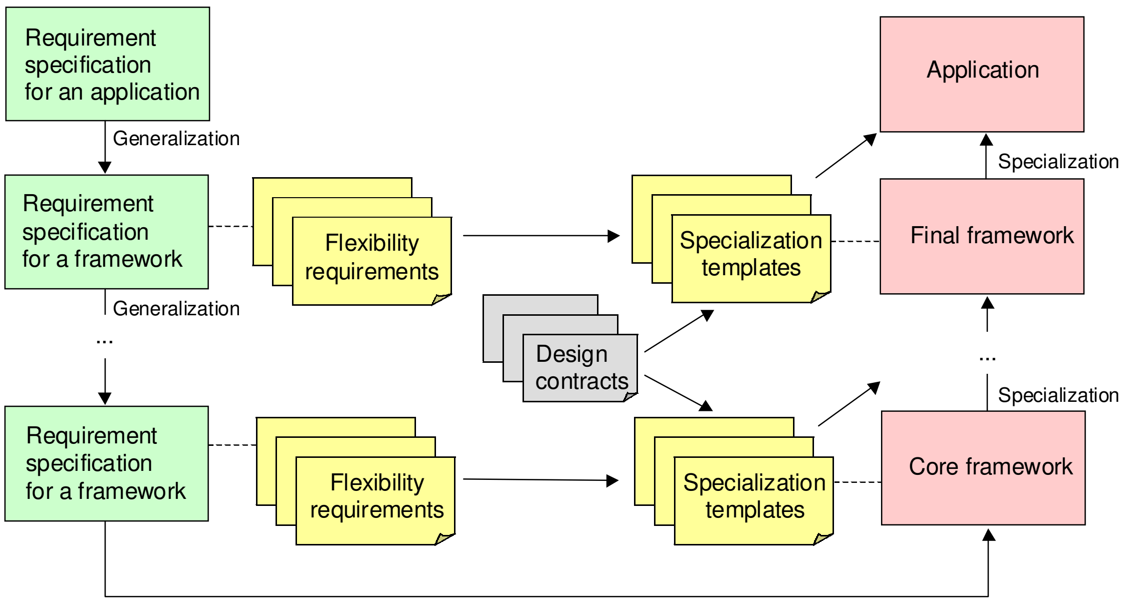


Figure 1. Flexibility requirements, design contracts and specialization templates in framework design based on stepwise generalization.

This design scheme for frameworks is idealistic in many ways, but we feel that it gives a useful guideline that the framework developer should have in mind, even though the scheme were not strictly followed. In particular, we argue that the concepts of a flexibility requirement and a specialization template are instrumental in framework design and documentation. However, note that the FRED tool described below does not care about how the specialization templates are produced.

4 FRED: Tool support based on specialization templates

FRED is a tool for managing framework-based software development. The basic idea is to use specialization templates (and design contracts) for active documentation of a framework. Our vision is that a framework can be specialized under the guidance of FRED, in the same sense as various kinds of wizards assist the programmer in the case of, say, GUI frameworks. However, FRED is completely domain-independent: it can be used to construct a "wizard" for any OO (Java) framework. The basic mechanisms of FRED take care of managing the bindings of specialization templates to actual source code, generating default implementations on the basis of example specializations, keeping track of the remaining tasks that have to be done, displaying informal instructions for specialization (e.g. specialization hints), enforcing the constraints defined in the specialization templates, and editing source code. A dedicated source code editor is integrated with the system so that all modifications of the source text are immediately checked against the specialization templates (and design contracts they are based on). A violation automatically generates a new task for the user in which this violation has to be removed.

The first stage in using FRED is to transform the design contracts into formal Cola specifications. The Cola compiler is currently under development; so far the Cola specifications have been manually transformed into the internal representation format of FRED.

The specialization templates are created through the graphical user interface of FRED. Within FRED, the specialization templates based on the known design contracts are instantiated, and some of the roles are bound to existing Java entities. FRED generates and automatically maintains a task list for the remaining unbound roles and checks that the bindings are legal. As a result, a collection of partially bound specialization patterns is created, representing the specialization interface of the framework.

Figure 2 gives an overview of using FRED. The user interface of FRED consists of structural views of design contracts (left below) and specialization templates with (partial) bindings (left above), a task list, an info sheet (right below) and a work area. The icon of an item (role) in a specialization template indicates the type of the item and whether it is bound or not. The standard tool used in the work area is a source editor.

However, various other tools for more high-level (and possibly domain-oriented) tools can replace the source editor, using a general tool interface.

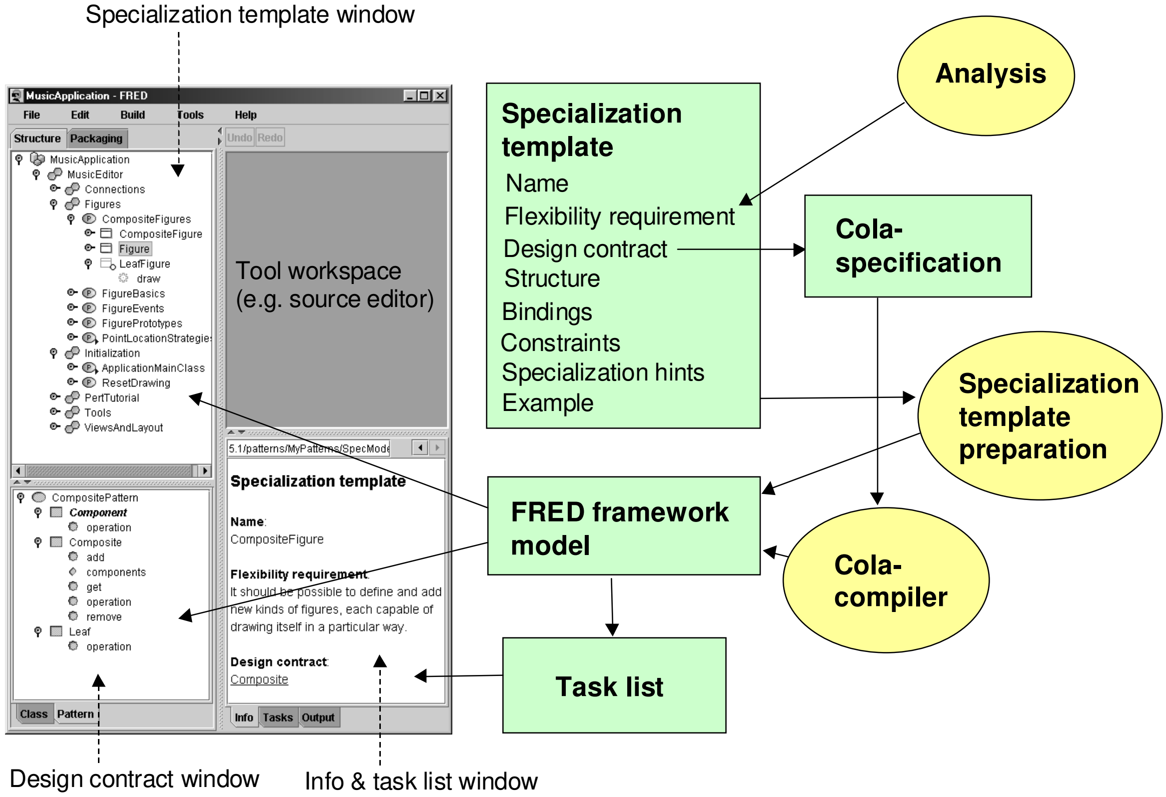


Figure 2. FRED in the framework specialization process.

In the user interface snapshot of Figure 2, the specialization templates have been created for JHotDraw, representing the specialization interface for this framework. One can immediately see that the hot spots in JHotDraw are related with connections (of figures), figures, initialization, tools (for manipulating figures and connections), and views/layout. For each template item in the specialization template view, the user can ask to see the (hierarchical) list of remaing specialization tasks. Each task item can be executed either by generating a default implementation or by binding some open role in the template to an actual Java entity. In the latter case the system checks whether the given Java entity conforms to the structural constraints given in the template (or actually in the design contract). One can also ask for the information sheet associated with the template; this is an informal hyperlinked (HTML) description of the template. When the task list is empty, the user can compile and run the resulting application through FRED as well.

5 Conclusions

Wrapping a framework up inside FRED yields several benefits. Constructing a framework becomes more systematic and manageable since FRED forces the designer to explicitly specify the specialization interface of the framework as specialization templates. This is particulary useful for white-box frameworks. Common design contracts can be specified and stored as reusable foundations of specialization templates. On the other hand, an application developer need not understand the framework as a whole: the specialization templates give condensed information about the relevant parts of the framework. The amount of information the user must perceive is relatively small[5], and FRED guides her through the specialization tasks by maintaining a things-to-do list. Further, FRED guarantees that the specializations follow the assumptions made by the framework developer, as described in the specialization templates and design contracts.

So far FRED is a fairly "syntactic" tool: it takes care of structural aspects of hot spots, rather than their semantics. A possible future approach is to allow the specification of methods in the design contracts, for example using pre- and post-conditions. However, introducing a full-fledged program verification system is beyond the scope of FRED: we wish to keep FRED as simple as possible. An attractive compromise might be to apply the idea of "grey-box components" introduced in [BW97] in which a certain pattern is specified for a method, rather than the black-box semantics (pre- and post-conditions).

References

[BW97] Büchi M., Weck W.: A Plea for Grey-Box Components. TUCS Technical Report 122, Turku Centre for Computer Science, August 1997.

[CACM97] *Communications of the ACM* 40, 10 (October 1997). Special issue on Object-Oriented Application Frameworks.

[Cas95] Casais E.: An Experiment in Framework Development. *Theory and Practice of Object Systems* 1, 4 (1995), 269-280.

[FMW97] Florijn G., Meijers M., van Winsen P.: Tool Support for Object-Oriented Patterns. In: *ECOOP 97*, LNCS 1241, Springer-Verlag 1997, 472-495.

[Gam98] Gamma E.: The JHotDraw framework. Download: http://members.pingnet.ch/gamma/JHD-5.1.zip.

[GHJV95] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley 1995.

[Hak98] Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J.; Pattern-Oriented Framework Engineering with FRED. In: Object-Oriented Technology, *ECOOP 98* Workshop Reader, Springer-Verlag 1998, 105-109.

[Hol93] Holland I.: The Design and Representation of Object-Oriented Components. PhD thesis, Northeastern University, 1993.

---

5 For example, for JHotDraw we needed 16 specialization templates.

[Joh92] Johnson R.: Documenting Frameworks using Patterns. In: *OOPSLA '92*, SIGPLAN Notices 27,10 (Oct 92), 63-76.

[KoM95] Koskimies K., Mössenböck H.: Designing a Framework by Stepwise Generalizaton. In: *Proc. of 5th European Software Engineering Conference* (ESEC '95), Sitges, Spain. LNCS 989, Springer-Verlag 1995, 479-498.

[Lew95] Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J., Schmucker K.: *Object-Oriented Application Frameworks*. Manning Publications/Prentice Hall 1995.

[MDE97] Meijler T.D., Demeyer S., Engel R.: Making Design Patterns Explicit in FACE - A Framework Adaptive Composition Environment. In: *Proc. of 6th European Software Engineering Conference* (ESEC '97). LNCS 1301, Springer-Verlag 1997, 94-111.

[Pre95] Pree W.: *Design Patterns for Object-Oriented Software Development*. Addison-Wesley 1995.

[Sch97] Schmid H.A.: Systematic Framework Design by Generalization. In: [CACM97], 48-51.

[SBF96] Sparks S., Benner K., Faris C.: Managing Object-Oriented Framework Reuse. *Computer* 29,9 (Sept 96), 52-62.