Task-Based Tool-Support for Framework Specialization

Markku Hakala
Tampere University of Technology
P.O.Box 553
FIN-33101 Tampere
Finland
markku.hakala@cs.tut.fi

ABSTRACT
An idea of providing assistance for framework-specialization by the means of interactive task list is presented. A model based on the notion of generative patterns is described, that provides basis for task-based tool support for framework specialization.

1 INTRODUCTION

Product line architecture is a system of rules and conventions for creating software products for a given domain. The broader the domain, the more complex rules the architecture embodies.

This growing complexity leads to a usability problem that can seriously hinder the reuse of the architecture. Thus, there is a need to promote to the comprehensibility of these rules, as well as validate the application of these rules.

Object oriented frameworks are an established way to implement product line architectures. The framework implements the invariant part of the architecture and defines a specialization interface. A new product can be derived from the framework by providing the application-specific part using the provided interface.

In frameworks, the features of the implementation language are used to express the architectural rules and conventions. This most often incorporates inheritance.

However, only fraction of the rules of the framework can be expressed adequately with language constructs. Every framework contains rules implied by the implementation but not statically checkable by the compiler.

This emphasizes the importance of the documentation. Thus, much effort has been placed on the research on framework documentation, as well as technologies and tools supporting the framework specialization.

This paper proposes a task-driven approach to framework specialization. The model embodies an algorithm that maintains a dynamic "things-to-do"-list in co-operation with the framework developer. The idea is to provide interactive specialization instructions that adapt to the current specialization problem.

In addition, code generation, and to some extent, verification, can be implemented on top of the model.

Proving the correctness of the specialization is however, beyond the scope of this research. Our approach aims to improve specialization process by emphasizing the education of the developer beforehand or concurrently with the process, rather than validation that usually takes place after the code has been produced.

This model can also be seen as an extension of the notion of Framework Cookbooks [Pre95]. However, the task driven approach has appeared to be useful also outside the scope of framework specialization, e.g. in formalizing architectural standards and conventions such as Java Beans.

The suggested model has been implemented in a development tool called FRED (Framework Editor). The tool has been evaluated in a realistic environment in the Finnish industry.

Chapter 2 describes the task driven programming model. Chapter 3 discusses the current implementation of the model along with a minimal case study.

2 TASK DRIVEN MODEL

Overview

Nowadays software development rarely starts from scratch. Each piece of software is produced against some standards, conventions and underlying systems and architectures. Most software structures manifest patterns that have been invented before. Everyone can agree that more than inventing ideas, software engineering is about applying ideas.

Applying an idea means following a plan to manifest that idea. The idea of task-based tool support is to model this plan as a list of tasks.

When dealing with complicated structures, which are typical for software, this task list cannot be adequately expressed by a linear step-by-step list. Making software is conceptually harder that making supper. A choice made during the process may change the rest of the plan completely. That is why cookbooks [Pre95], although a step to a right direction, are not enough.

By doing a task, the developer continues with the plan, but also makes choices. There might be several ways to do the task, each of which will generate succeeding tasks.

Technically, there could be two ways to do tasks. The developer may either explicitly mark a task as done, or the tool may implicitly consider a task as done.

The problem with traditional documentation is that it has to be written before the specialization takes place. Therefore the documentation has to communicate with the abstract concepts of the framework, not with the concrete concepts of the specialization. By providing tasks incrementally, the tool can gather information of the specialization and parameterize the documentation with the specialized concepts. The same information can be used to provide more active assistance in for task, e.g. code generation that adapts to the specialization context.

Pattern Based Model

In order to provide task-based tool support, we suggest a model that builds on the notion of patterns as generative descriptions that can be used systematically to produce a number of co-similar structures.

This fits a more traditional way to see a pattern as a description of a recurring problem along with a reusable solution to that problem within a certain context. However, our notion of patterns should not be confused with design patterns [Gam95], often associated with strict rules on their known uses and the domain of applicability. With the goal of providing programming assistance using patterns in mind, everything is a considered a pattern that contains a structure definable in a form described in this chapter. Most design patterns seem however to fall into this category. Nevertheless, considering our current model, tool support benefits most the instantiation of less abstract implementation-oriented variants of design patterns.

One essential part of a pattern definition is the description of the pattern structure. Within the scope of this research, we concentrate on the structure and leave the other parts open. However, a pattern is not seen as a purely declarative description of the solution, but rather a description of the process to instantiate the abstract solution. A pattern can therefore be seen as an algorithm that can be applied in several environments to build the same structure.

Given this informal description, the structure of a pattern can be presented as a directed acyclic graph (DAG) that can be formalized by the following 4-tuple:

$$P = (R, D, c, S)$$

This is called the *definition graph*. The vertices R of the graph are called *roles*, and the directed edges D are called dependencies. A dependency from r to s is an ordered pair (r, s), where role r is called the *depender*, and role s is called the *dependee*.

Function c is called the *cardinality function*, and is defined as a mapping from a role to an integer range. The range is defined by its end points and is called the *cardinality constraint* of the role. The cardinality function is defined as follows:

$$c : R \rightarrow \{0,1, \ldots, n\} \times \{1,2, \ldots, \infty\}$$

$$c(r) = (l, u), \ \ u \geq l$$

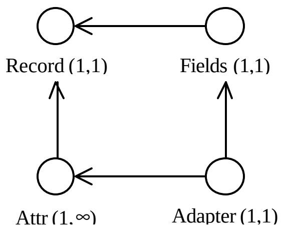
The definition graph represents a reusable abstract structure. The roles define all the abstract components of this structure, whereas the dependencies and cardinality constraints define the abstract relationships between roles.

The fourth part of the pattern graph definition is the *structural constraint set S*, which is a set of constraints that are used to restrict the instantiation process. Different kind of formalisms could be adopted for such constraints. E.g. predicate logic could be used to state the required properties. We are currently researching different alternatives in expressing structural constraints. One that has proved to be very useful in practice is what we call the *path constraint*. A path constraint C is defined as a set of paths[1], which all begin at the same role and end to the same role, i.e.:

$$C = \{ (r_1, \ldots, r_n) \mid (r_i, r_{i+1}) \in D \}$$

$$\forall \ (r_1, \ldots, r_n), (s_1, \ldots, s_m) \in C : r_1{=}s_1, r_n{=}s_m$$

Figure 1 presents an example pattern structure with a graphical graph notation, where nodes represent roles and edges represent dependencies. The roles are labeled, and the cardinality constraint is placed after each role label. Below the graph there is a path constraint between roles Adapter and Record. The same pattern is examined in detail by going through the specialization example in Chapter 3.



{(Adapter,Fields,Record), (Adapter,Attr,Record)}

Figure 1. An example pattern definition graph

Applying a pattern is called *instantiation*. It results in a *pattern instance*, which describes a concrete structure as an instance of the abstract structure defined by the pattern. In practice, this correlation is described by binding concrete *program elements*, like classes, methods, formal parameters and so on, to the abstract roles defined by the pattern.

At any time during the instantiation, the structure of the instance of pattern $P = (R, D, c)$ can be described as a DAG as well. It can be presented by a 3-tuple as follows:

$$P' = (R', D', s)$$

[1] A path within a DAG from vertex v_1 to vertex v_n connected with number of edges is defined unambiguously as an ordered list of vertices $(v_1, \ldots, v_n)$ on that path.

In this *instance graph*, the vertices *R'* are called *role instances*. Each role instance is a manifestation of some role in the definition graph. Function *m* called *meta-function* defines this relationship as a mapping from a role instance to a role:

$$m : R' \rightarrow R$$

Similarly, the directed edges *D'* between role instances are called *dependency instances*. Each dependency instance (*x*, *y*) manifests a dependency defined between the two roles that the role instances *x* and *y* are instances of. In other words:

$$\forall\ (x, y) \in D' : \exists\ (m(x), m(y)) \in D$$

The last component of the instance graph, function *s*, called *state function*, maps each role instance to a *state*. The function is defined as follows:

$$s : R' \rightarrow \{mandatory, optional, valid, invalid\}$$

To understand the state function, we must look at the instantiation process.

The instantiation process is incremental and guided by the definition graph of the pattern. The appendix lists an algorithm that takes a definition graph *P* as an input, and gradually constructs an instance graph *P'* and function *m*. The algorithm assumes a development tool environment that works in the interaction with the developer.

The algorithm works by gradually augmenting the instance graph with new role instances. New instances are generated based on the dependency and cardinality constraint definitions. Each new role instance is considered a task to be carried out by the developer. A state, either *mandatory* or *optional* (with obvious semantics), is assigned to the task. Once the user does the task, the state of the role instance is changed. The state becomes *invalid*, if the tool is able to judge that the task is not properly executed, or *valid* in other cases.

In addition, whenever a task becomes done, the algorithm may augment the pattern instance with new role instances.

Mandatory, *optional* and *invalid* role instances are called as *tasks*. *Valid* role instances constitute the concrete structure that has been instantiated from the abstract structure defined by the pattern. If the definition graph is well formed, and the user follows the ever-changing task list, a point is reached where all role instances are either *valid* or *optional*. As a result, an abstract structure defined by the pattern has been specialized in a user-defined context.

Figure 2 presents a partial pattern instance and its relation to a pattern. The state is indicated by a superscript after role instance label and the meta-function is indicated by edges between role instances and roles. Path constraints are omitted from this figure.

The definition graph provides syntax for describing patterns. To provide useful tool support, the graph has to be decorated with tool-specific semantics. Therefore, a practical application of this model extends it in many ways. E.g., tool-specific attributes could be defined for the roles. Such attributes could include the default implementation for the required program element, documentation, programming-language specific constraints, such as the return type of the method or required inheritance, and so on. Provided sufficient information exists, the tool can generate context-specific instructions for the task, provide automation such as code generation, and validate the user actions.

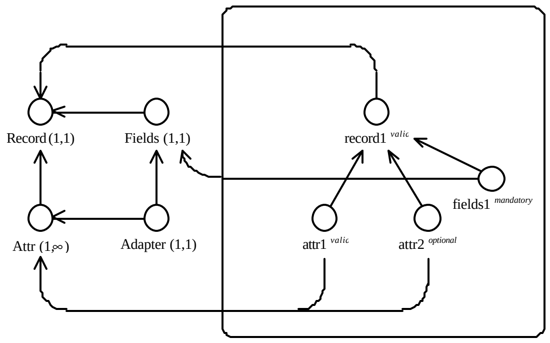


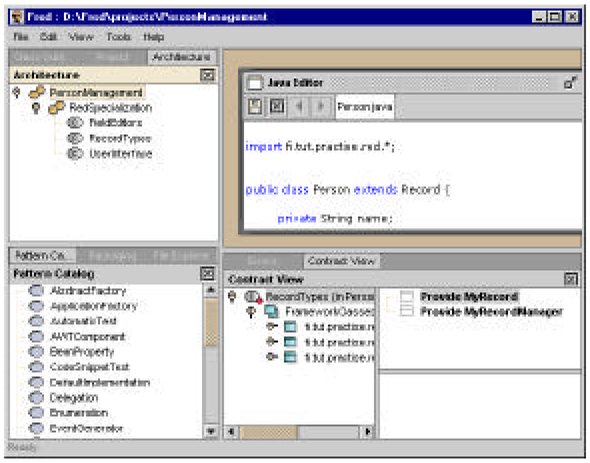
Figure 2. An example pattern instance

The tool-specific attributes may also include rules and heuristics to determine automatically when the user completes a certain task. In practice however, the tool cannot rely on such heuristics alone. Instead, the user should explicitly *acknowledge* some tasks. By requiring explicit commitment from the user, the tool may behave in a more predictable way. E.g., if the user is converting one of his classes to a Singleton [Gam95], the tool may provide assistance much earlier and reliably if the user explicitly states his or her intentions.

Completing a task may require "evidence" to be provided by the user. In a development tool, such evidence could be part of the produced source code. This program element can then be checked against the rules imposed by the pattern and any inconsistencies may be reported. This equals to enhancing the compiler with architecture-specific typing checks. Ideally, the tool provides an incremental development environment where these checks could be re-evaluated whenever the user manipulates the source code, thus making it possible for the task-list to evolve by itself concurrently with the development process.

3 FRED AND FRAMEWORK SPECIALIZATION

FRED is a prototype tool implementing the discussed model. FRED is implemented in Java and intended for providing task-driven assistance for Java programming, especially to support specialization of Java frameworks. The tool currently being tried out in Finnish software industry. A small snapshot of the user interface is shown in Figure 3.



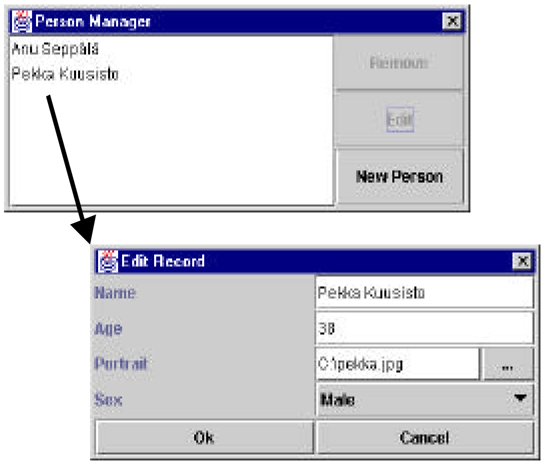
```
import fi.tut.practise.red.*;

public class Person extends Record {

    private String name;
```

Provide MyRecord
Provide MyRecordManager

Figure 3. User interface of FRED tool

As an example of task-driven framework specialization, a framelet called Red is used. It comes with FRED 1.1 release. Framelets [Prk98] are small frameworks consisting a handful of classes and used as reusable building blocks for creating components. The Red framelet is an evolved version of a framelet discussed in [Prk98].

Red provides user interface facilities to maintain a list of Record-objects and edit their fields. A specialization of Red typically defines a new Record subclass with some application domain –specific fields. Once the user has defined this new record type and derived some other classes, the framelet can generate the user interface automatically. Typical user interface windows provided by Red are seen in Figure 4.



Anu Seppälä
Pekka Kuusisto

Name	Pekka Kuusisto
Age	38
Portrait	C:\pekka.jpg
Sex	Male

Figure 4. Typical views provided by Red framelet

Through this example, a development tool similar to FRED environment (see acknowledges) is assumed, that guides in the instantiation process.

Figure 1 presented a simple pattern definition graph, without any tool-specific attributes and semantic constraints. The graph describes a pattern with roles named *Record*, *Attr*, *Adapter* and *Fields*. Let us assume that sufficient information, although suppressed from our simplified notation, is provided for the tool to interpret *Record* as a description of a class role, *Attr* as a role for class member variable, *Fields* as a method role, and *Adapter* as a role that describes some piece of code to be written within a method. This minimal pattern is a simplification of a pattern enclosed in the FRED release, to describe creation of a new Record subclass using Red framelet.

Given this description of the pattern, the tool can provide a task list for the user. The tool begins by instantiating each role that has no dependencies. Therefore, an instance of role *Record* is created. Let us name it *record1*. The state of the instance is set to *mandatory*, which denotes a mandatory task. This task tells the developer to provide a class to play the role *Record*. Provided the proper semantic constraints are included, this task means that the developer should subclass the Record-class provided by the framework. Based on the tool-specific attributes associated with the role, most notably the free-form documentation, the tool can generate textual description for the task as well as suggest a skeletal implementation.

This is the first and only task the tool is able to generate at this point. This instance graph at this point is presented visually in step 1 of Figure 5.

To complete the task, the user has to point out a suitable class for the tool. This may involve creation of a new class, modification of an existing one, or letting the tool generate the skeletal default implementation. Suppose the developer is specializing Red to store information on personnel. For that purpose, the developer creates a new Record subclass named Person. If heuristics are provided for the pattern to determine every subclass of Record to play the role *Record* in the pattern, the tas k may be considered done automatically. If such heuristics are unavailable, the user must point out the Person class explicitly to adhere the task.

The Person class may be left empty for the moment. If the inheritance constraint is satisfied, the tool changes the state of the role from *mandatory* to *valid*. The binding between the class and the role instance is maintained, so that the state can be re-evaluated whenever there are any changes in the associated source code.

After completion of the first task, the tool re-evaluates the pattern instance against the definition. No more instances of role *Record* are created because the upper bound of the associated cardinality constraint is 1, and an instance of *Record* already exists. However, an optional task named *attr1* to create a member variable to play role *Attr* is created, along with a dependency instance (*attr1*, *record1*). The task is mandatory, as the lower bound of the associated cardinality constraint is 1, requiring there to be at least one instance of *Attr* for each instance of *Record*. Similarly, an instance of *Fields* is created. This task corresponds to overriding a method declared in the Record-class. Step 2 in Figure 5 presents the instance graph at this point.

Given these two tasks, the user may continue in his or her preferred order. Note that the graphical representations do not describe any semantics on the structure. Such semantics, including the requirement for the expected fields and methods to be declared in the class for the associated instance of *Record*, has to be defined by tool-specific extensions to the model.

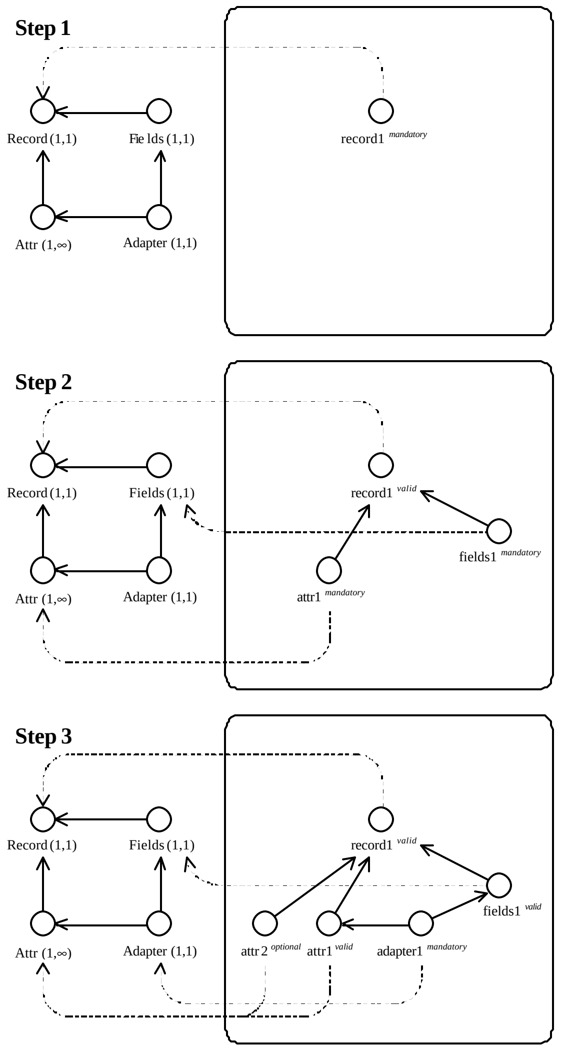


Figure 5. Some specialization steps for specializing Red

For task *attr1*, the developer introduces a new member variable in the Person class, and denotes this variable as the required program element. For *fields*1, the developer asks the tool to generate the default implementation. Step 3 of the same figure presents the situation where the developer has done both these tasks and the tool has re-evaluted the pattern instance once again. Two new tasks have been created. One concerns creating another member variable and is optional. The other instructs the developer to type in some adapter code in the overridden method. In Red specialization, this adapter code provides access to read and write the member variable through the Red user interface. Our pattern definition states that such adapter code must exist for each member variable declared to play the role *Attr*.

A mechanism can be provided to undo tasks, providing the means to backtrack the instantiation process and reconsider the decisions made. In FRED, the source code is modified under supervision of the tool, thus earlier decisions are refined automatically based on the modified source code. Whenever the code no longer complies with constraints of the pattern, the associated role instances become invalid, reminding the user of the architectural rules and conventions.

ACKNOWLEDGEMENTS

The model on task-driven programming has been developed within FRED project, between the Department of Computer Science at the University of Helsinki and the Department of Computer and Information Sciences at the University of Tampere, funded by TEKES and several industrial partners. The work is being continued in the Department of Information Technology at the Tampere University of Technology, Finland. Earlier work is reported in [Hak99]. FRED tool can be downloaded from the project web site at http://practise.cs.tut.fi/fred.

REFERENCES

[Gam95] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns – Elements of Reusable Object-Oriented Architecture*, Addison-Wesley, 1995.

[PrK98] Pree W., Koskimies K.: Framelets – Small and Loosely Coupled Frameworks. Manuscript, submitted for publication, 1998.

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

[Hak99] Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J., Koskimies K., Paakki J.: Managing Object-Oriented Frameworks with Specialization Templates. *ECOOP'99 Workshop on Object Technology for Product-line Architectures*, 1999.

APPENDIX: PATTERN IN STANTIATION ALGORITHM

The following algorithm *UPDATE* expects a pattern $P = (R, D, c, S)$ and a pattern instance $P' = (R', D', s)$. In addition, it is assumed that all structural constraints in set S are path constraints. The algorithm makes also use of the following definitions:

```
$I(r) = \{ x \mid m(x) = r \}$
$dependees(r) = \{ s \mid (r, s) \in D \}, r \in R$
$x \rightarrow r = y, y \in I(r), r \in R, (x, y) \in D'$
```

The algorithm should be evaluated by a development tool to update P' and the meta-function m, that defines the mapping between P' and P. Provided the state function returns *valid* or *invalid* for tasks that have been done at that time, the algorithm may construct new tasks with either *mandatory* or *optional* state. Modification of a function or set is denoted by a left-pointing-arrow "$\leftarrow$".

```
UPDATE is

  For each $r \in R$ Do

      $\{ s_1, ..., s_n \} \leftarrow dependees(r)$
      $(l, u) \leftarrow c(r)$

      For each $(d_1, ..., d_n) \in I(s_1) \times .. \times I(s_n)$
          where $\forall C \in S : \forall (r, a_1, ..., a_p), (r, b_1, ..., b_q) \in C : \exists i, j :$
              $d_i \in I(a_1)$ and $d_j \in I(b_1)$ and $d_i \rightarrow a_2 \rightarrow ... \rightarrow a_p = d_j \rightarrow b_2 \rightarrow ... \rightarrow b_q$ Do

          $X \leftarrow \{ x \in I(r) \mid \forall i \exists (x, d_i) \in D' \}$

          If $|X| < l$ and not $(\exists x \in X : s(x)=mandatory$ or $s(x)=optional)$ Then
              Let x = new role instance
              $m(x) \leftarrow r$
              $X \leftarrow X \cup \{ x \}$
              $R' \leftarrow R' \cup \{ x \}$
              $D' \leftarrow D' \cup \{ (x, d_1), ..., (x, d_n) \}$
          End

          If $\exists x \in X : s(x)=mandatory \vee s(x)=optional$ Do
              If $|X| \leq u$ Then $s(x) \leftarrow mandatory$
              Else $s(x) \leftarrow optional$
          End
      End
  End
```