Reverse Engineering Framework Reuse Interfaces

Jukka Viljamaa
Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland
+358 9 191 44506
jukka.viljamaa@cs.helsinki.fi

ABSTRACT

Object-oriented application frameworks provide an established way of reusing the design and implementation of applications in a specific domain. Using a framework for creating applications is not a trivial task, however, and special tools are needed for supporting the process. Tool support, in turn, requires explicit specification of the *reuse interfaces* of frameworks. Unfortunately these specifications typically become quite extensive and complex for non-trivial frameworks. In this paper we discuss the possibility to *reverse engineer* a reuse interface specification from a framework's and its example applications' source code. We also introduce a programming environment that supports both making and using such specifications. In our environment, the reuse interface modeling is supported by a *concept analysis* based reverse engineering technique described in this paper.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments – *integrated environments*. D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *reverse engineering*. D.2.13 [**Software Engineering**]: Reusable Software – *reusable libraries, reuse models*.

General Terms

Design, Documentation, Experimentation, Languages.

Keywords

Documentation, formal concept analysis, framework, pattern, reuse, reverse engineering.

1. INTRODUCTION

Reuse is one of the key factors in increasing the quality and productivity of software development [1, 23]. Accordingly, lots of resources have been invested in designing and implementing various kinds of reusable assets. Of these assets *object-oriented application frameworks* have proven to be especially cost-effective because they enable reuse of both functionality and architecture of systems [5, 10].

A framework captures the commonalities of its application domain by a set of carefully designed abstract classes. The relationships between these core classes and the algorithms implemented within them define the common architecture and functionality of the applications to be derived from the framework.

For an application developer the most crucial part of a framework is its *reuse interface*. The reuse interface consists of variation points or *hot spots* [21] related to specializing the framework's abstract classes (*specialization interface*) and calling or combining its concrete default components (*call interface*).

Using a framework through its specialization interface is often referred to as *white-box reuse* because it usually requires detailed knowledge of the framework's internal structure (e.g. how and when overridden methods are called by the framework) while *black-box reuse* only involves calling the framework's services through the interfaces of the concrete components. Similarly, frameworks that emphasize subclassing and method overriding are called *white-box frameworks* and frameworks that are mostly used by instantiating, combining, and configuring default components are called *black-box frameworks*.

Most framework research has concentrated on framework design and construction, whereas framework usability has been studied less even though it has lots of practical significance [26]. Frameworks are hard to use because they tend to be large, complex, and abstract systems. It has been estimated that it takes nearly a year for a professional programmer to master a large framework in order to be able to use it efficiently [2, 8].

It is already widely accepted that tool support is effective in the specialization of some particular kinds of frameworks. For example, *visual builders* have been used for constructing user interfaces for quite some time, and there are similar black-box framework tools for some other domains, too [6]. However, the framework usage problems are most severe with white-box frameworks that are extensively applied in the industry because they are generally more flexible than black-box frameworks. Since it is not likely that all (or even most) white-box frameworks would ever evolve to become (pure) black-box frameworks it is essential to provide systematic methods and tool support for assisting white-box framework usage, too.

In our vision, both black-box and white-box frameworks are accompanied with a programming environment that guides and controls application programmers in creating applications according to the conventions of the framework. In practice, the tool should offer (1) context-sensitive documentation that dynamically adjusts to the choices the developer makes, (2) parameterized code generation to automate the production of skeletal implementations, and (3) validation of application code against the architectural requirements of the framework.

The most important prerequisite for that kind of tool support is the existence of a precise notation for expressing framework reuse interfaces. In this paper we will briefly introduce such a notation. After that we will concentrate on applying reverse engineering techniques to automatically extract framework reuse interface specifications from the source code of the framework itself and from any available samples of its specializations.

We proceed as follows. First, in Section 2 we look at some methods that have been proposed for solving the framework usage problem. Section 3 introduces our model for specifying framework reuse interfaces. In Sections 4, 5, and 6 we present a new technique for reverse engineering framework reuse interface specifications from implementation. In Section 7 we describe a case study and summarize our experiences with a prototype implementation of our technique. Section 8 concludes the paper and gives some directions for further work.

2. RELATED WORK

Many authors emphasize the role of example applications in learning frameworks [10, 17, 26]. An example application demonstrates how the framework's classes can be instantiated, specialized, and used in an application. Examples are concrete and thus easier to learn than the abstract framework in itself and programmers are usually quite skilful in inferring framework usage protocols from them. However, the examples by themselves are not enough to support the construction of more evolved and complex specializations. That is why framework reuse interfaces must be systematically documented.

When an application developer wants to use a framework to solve a problem it should be easy for her to find a solution and to apply the solution to her problem. To achieve this kind of accessibility, framework documentation can be organized as a *framework cookbook* [18]. Each entry in a cookbook is a recipe that describes a (common) problem and gives set-by-step guidance for solving it.

Reuse documentation in the form of cookbooks is quite constraining, however. Cookbooks describe only a limited set of predefined ways of reusing a framework [3]. Instructions for adapting a framework cannot be adequately expressed as a static and linear step-by-step task list, because a choice made during the specialization process may change the rest of the list completely. Furthermore, framework cookbooks rely on narrative descriptions that may be imprecise or incomplete.

Many attempts have been made to make the cookbook approach to framework documentation more dynamic and more precise (see, e.g., *hooks* [9] and *SmartBooks* [19]). The common denominator in all these approaches is to define a language for representing the reuse interface of a framework. Specifications written in that language can then be dynamically interpreted to provide task-based assistance for framework specialization.

A *design pattern* describes a recurring solution to a common design problem [13]. Often the purpose of the pattern is to introduce variability into a system in order to make it more reusable or extensible. Many of the well-known design patterns are drawn from the experiences gained while designing application frameworks. It is therefore no wonder that they are especially effective in framework documentation [17].

The design pattern's structure works as a template that outlines roles, responsibilities, and relationships for the program elements (e.g. classes, methods, or fields) that participate in the solution. *Role-based models*, such as design patterns, are known to be suitable for describing reusable software systems (see, e.g., [13] or [14]).

We feel that framework specialization instructions can most conveniently and intuitively be described in terms of role-based models. Our goal has been to combine the intuitive task-driven framework assistance provided by cookbooks, the concreteness of example applications, and the precise, declarative nature of role-based pattern formalisms. In the following we introduce our own simplified role-based framework reuse interface specification language. However, we argue that similar constructs are found in almost all role-based languages and that the reverse engineering technique described in Sections 4, 5, and 6 can be applied regardless of the exact target language or model.

3. SPECIFYING REUSE INTERFACES

Assisting framework specialization with a tool requires an explicit and precise mechanism for expressing framework reuse interfaces. The reuse interface of a framework typically involves complex requirements and restrictions among multiple program elements. Such relationships are difficult or even impossible to express using the current implementation languages[1]. Thus, we have two choices: we can either design a new implementation language (extension), or we can construct a tool that uses a separate specification for enforcing those restrictions that are not directly supported by the constructs of the framework's implementation language.

Even though a new language might be considered a more elegant solution to this problem, there are also quite a few advantages to the solution that utilizes a separate specification. First of all, when the specification language is separate from the actual implementation language, it is possible to use an existing (popular) implementation language. It is also relatively easy to adjust the method to fit multiple programming languages. Furthermore, the tool itself can be integrated into an existing development environment so as to get full advantage of an established environment and the standard tools it provides.

Using a separate specification makes it possible to model existing frameworks without modifying their implementations. It is also feasible to make multiple reuse interface specifications for different user groups. For example, novice users might have a simplified and restricted view to a framework, whereas a reuse interface model for experts might reveal more details and advanced features. Also, from a conceptual point of view, separate reuse interface specifications provide a more abstract and high-level view to the framework than possible language extensions do.

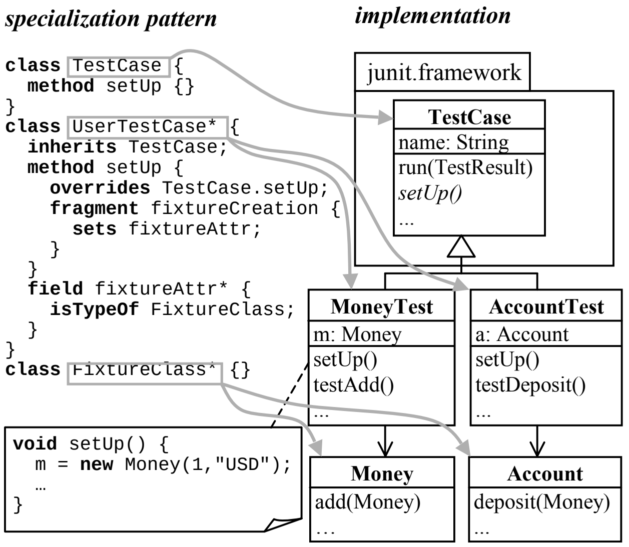
In the following we will use *specialization patterns* for specifying the reuse interface of a framework [16]. A specialization pattern is

[1] There are mechanisms that enable the prevention of some simple framework misuses. In Java, for example, it is possible to declare classes and methods *final* to distinguish the framework's *frozen spots* from the hot spots.

a specification of a program structure, which can be instantiated in several contexts to get different kinds of concrete structures. It is given in terms of roles to be played by structural elements of a program.

A role is always played by a particular kind of a program element. Consequently, we can speak of class roles, method roles, field roles and so on. For each kind of a role, there is a set of constraints that can be associated with the role. For instance, for a class role there can be an inheritance constraint specifying the required inheritance relationship of each class associated with that role. All roles have a cardinality that bounds the number of program elements that can play the role. The cardinality is given with respect to the other roles the role refers to in its constraints.

Figure 1 shows a simplified example of a specialization pattern (on the left) that models a set of program elements (the UML class diagram on the right). The pattern (partially) specifies the hot spot for setting up test fixtures in the JUnit testing framework [11]. It says that if the developer wishes to have a common test fixture for all test scripts defined in a test case, she must bind her *TestCase* subclass to the *UserTestCase* role and then provide a method, which overrides the *setUp* method declared in *TestCase*. In a concrete test case there can be zero or more (as indicated by the cardinality symbol '*' after the role name) fixture attributes for each fixture class selected by the developer. For each of those fixture attributes there must be a code fragment within the *setUp* method that initializes it by assigning an appropriate object to it.



```
class TestCase {
  method setUp {}
}
class UserTestCase* {
  inherits TestCase;
  method setUp {
    overrides TestCase.setUp;
    fragment fixtureCreation {
      sets fixtureAttr;
    }
  }
  field fixtureAttr* {
    isTypeOf FixtureClass;
  }
}
class FixtureClass* {}
```

```
void setUp() {
  m = new Money(1,"USD");
  …
}
```

Figure 1. A specialization pattern with its roles bound to program elements

Here we omit the details of the specialization pattern language and its interpretation in tools. However, the dotted arrows in Figure 1 pointing from the roles on the left to the associated program elements on the right illustrate how a pattern can be bound to a piece of source code to check whether or not that code adheres to the restrictions specified by the roles and their constraints[2]. A pattern can also be used to generate code and provide task-based assistance for framework specialization as described in [15] and [16].

[2] For simplicity, only class role bindings are shown in the figure.

4. REVERSE ENGINEERING USING FORMAL CONCEPT ANALYSIS

Reverse engineering is a discipline that tries to extract high-level descriptions of a system from its source code to supplement or replace missing or outdated documentation [4, 22, 31]. In the context of framework reuse interface modeling, reverse engineering techniques, especially *design recovery*, can be used to recover valuable information about the hot spots of a framework by using the source code of the framework itself and a set of available example applications as input for the analysis.

4.1 Library-Based Design Recovery

In design recovery the goal is to locate instances of abstract design artifacts, such as design patterns, in the source code of a system. A common way to identify abstract design artifacts is to use a predefined library of design artifact descriptions expressed in a suitable formal language (e.g. in PROLOG as is done in Maisa [20]). Source code structures can then be matched against the library to find instances of the abstract design artifacts.

In practice, the library-based approach to design recovery cannot guarantee complete understanding of the target system, because every non-trivial system always contains some code that is idiosyncratic [22, 28]. Thus, the success of any library-based method depends on how much of the code that is being examined is stereotypical and can be analyzed by matching against a library of design artifact descriptions. Furthermore, constructing such libraries requires a lot of effort and if one decides to use an existing library, one is faced with the problem of choosing an appropriate library for the current situation.

Frameworks are often reported to contain instances of well-known design patterns. Unfortunately, according to our experiences, many of the found pattern instances are structurally framework-specific variants. This means that if the abstract design pattern descriptions stored in the library are relatively detailed they rarely exactly match pattern instances in frameworks. To acquire more matches the constraints in the pattern descriptions can be relaxed, but then the acquired information will also be less precise and, furthermore, the risk of getting false positives increases. The other possibility is to customize the library to give framework-specific descriptions of the variations to accurately match them to the actual pattern instances in the framework implementation. But that, of course, would require detailed knowledge of those instances before the analysis. For these reasons the library-based solution, at least in a general case, does not work for locating framework hot spots.

4.2 Formal Concept Analysis

Instead of library-based design recovery techniques, we propose using *formal concept analysis* (FCA) [12] to extract role-based specialization instructions from the available source code. Formal concept analysis is a general mathematical method for identifying commonalities within systems. It has been successfully applied for semi-automatic modularization of software systems [24, 25] as well as for detecting instances of design patterns without a predefined pattern library [28]. Other work using FCA for inferring high-level information about software systems include identification of code configurations [27] and program features [7].

In general, FCA provides a way to discover sensible groupings of *objects*[3] that have common *attributes* in a certain *context* [24]. Informally, a *concept* is a collection of all the objects that share a set of attributes in a given context. The set of common attributes of the concept is called the concept's *intent* and the set of objects belonging to the concept is called the concept's *extent*.

As an example of a context, think of various kinds of sports as objects and certain characteristics of those sports as attributes. That context could be expressed as a table showing which characteristics each sport has (see Table 1).

Table 1. A simple context of sports and their characteristics

		attributes (A)			
	R	*athletics*	*ballgame*	*team sport*	*Olympic sport*
objects (O)	*bowling*		√		
	cricket		√	√	
	javelin	√			√
	long jump	√			√
	tennis		√		√
	volleyball		√	√	√

Formally a context is a triple $C = (O, A, R)$, where O and A are finite sets of objects and attributes, respectively, and R is a binary relation between O and A. Let $X \subseteq O$ and $Y \subseteq A$. Let us also define mappings $\sigma(X) = \{a \in A \mid \forall o \in X\colon (o, a) \in R\}$ (the common attributes of X) and $\tau(Y) = \{o \in O \mid \forall a \in Y\colon (o, a) \in R\}$ (the common objects of Y). Using these definitions a concept in the context C can be defined as a pair of sets (X, Y) such that $Y = \sigma(X)$ and $X = \tau(Y)$, where X is the concept's extent and Y is the concept's intent. For example, ({*cricket*, *volleyball*}, {*ballgame*, *team sport*}) is a concept in the context given in Table 1.

5. FORMING CONTEXTS FROM CODE

In order to extract a role-based specification for a hot spot of a framework we can select relevant program elements and their properties from both the framework itself and its available specializations and use them as concept analysis objects and attributes (see Figure 2). When a context has been formed, a simple algorithm can be used to produce concepts from the context. After that, a suitable subset of those concepts is chosen and the extent of each selected concept is translated to a role and the intent of the concept is translated to a set of constraints for that role.

The extraction process can be iterated to declare subroles under each role acquired during the previous iteration cycle (e.g. to declare method roles within class roles). The number of iterations depends on the level of detail present in the input.

To ensure the effectiveness of the method, the input should include a representative selection of all possible framework specializations. In principle, the input can be given at any level of precision. It should be noted, however, that the amount of detail present in the input dictates the precision of the analysis and the accuracy of the extracted patterns. For example, to produce specialization patterns that include default implementations for method bodies, the input must contain information about the method implementations found in example applications.

[3] "Objects" of concept analysis shall not be confused with "objects" of object-oriented programming.

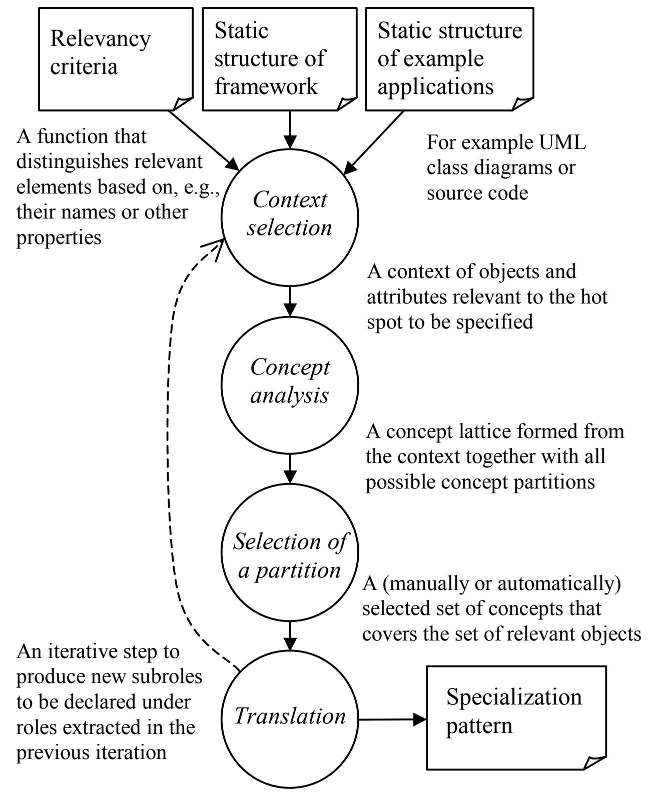

Relevancy criteria
Static structure of framework
Static structure of example applications
A function that distinguishes relevant elements based on, e.g., their names or other properties
For example UML class diagrams or source code
Context selection
A context of objects and attributes relevant to the hot spot to be specified
Concept analysis
A concept lattice formed from the context together with all possible concept partitions
Selection of a partition
A (manually or automatically) selected set of concepts that covers the set of relevant objects
An iterative step to produce new subroles to be declared under roles extracted in the previous iteration
Translation
Specialization pattern

Figure 2. Extracting a specialization pattern from implementation

5.1 Selecting Program Elements as Objects

The structure of the concept analysis context (i.e. objects and attributes) that is to be formed from the input depends on the kind of roles that are to be produced. For example, when extracting class roles, the classes and interfaces present in the input will be selected as objects and their features (e.g. inheritance relationships, declared methods, and data fields) will be selected as attributes.

In general, for each role kind we assume a set of program elements from which the objects will be selected and a set of *property functions* that dictate the attributes. Attributes are defined by applying each property function to each element and by collecting all distinct results as the set of attributes.

To be more precise, let us define a program element e as a pair $(k, n) \in E$, where $n \in N$ is a name that belongs to the set of unique element names (N), identifying the element among the set of all elements (E) and $k \in K$ is element kind that belongs to the set of all possible element kinds (K). For instance, for a Java system E_{Java}, the set of element kinds K_{Java} = {*class*, *method*, *field*, …}. Similarly, the set of names N_{Java} could include the path names of all elements in E_{Java}. We usually refer to an element by its name and omit its kind if there is no danger of confusion.

A property function $f\colon E \rightarrow \wp(E)$ is a mapping from a program element to a set of (other) program elements. Let $E_k \subset E$ be a set of all program elements of kind k. The element kind (e.g. *class* if k

$\in K_{Java}$) determines a set of property functions F_k that are applicable to the elements of that kind, i.e. $F_k = \{f \mid f\colon E_k \to \wp(E)\}$.

Given an element $e_k \in E_k$, each property function $f \in F_k$ yields a *property value* $V = f(e_k)$. The property value V is often a set that contains only one element (e.g. in case f is a mapping from a method to its return type), but it can also contain multiple elements. For example, for Java classes the set of property functions could be defined as F_{class} = {*inherits*, *declares*}, where *inherits* maps a class to the set of classes and interfaces it extends or implements and *declares* maps a class to the set of methods and fields it declares. Finally, let us define a *property* p_e of an element $e_k \in E_k$ as a pair (f, $V = f(e_k)$), where $f \in F_k$ and $V \in \wp(E)$.

The properties that the property functions determine when they are applied to each program element at a time will be represented as attributes in the context that will be built. For instance, consider the following source code fragment as input for extracting class roles (of Figure 1):

```
class TestCase {
  void setUp() {}
}
class MoneyTest extends TestCase {
  void setUp() { … }
}
class AccountTest extends TestCase {
  void setUp() { … }
}
```

Suppose that the property functions to be considered when producing class roles include *inherits* and *overrides*, and that they return the inherited base class of a class and the methods that are overridden in the class, respectively. In addition, let us assume that a special attribute *name* is introduced for those classes that do not have any other attributes determined by the general property functions (in this example *TestCase* does not inherit any classes or override any methods). With these rules for producing attributes with property functions we can form a context for determining the class roles based on the input given above (see Table 2).

Table 2. A context for determining class roles

	R	*attributes (A)*		
		name TestCase	*inherits TestCase*	*overrides setUp*
objects (O)	*TestCase*	√		
	MoneyTest		√	√
	AccountTest		√	√

5.2 Producing a Concept Lattice

It is possible to mechanically determine the concepts of a given context [25] and to form a *concept lattice* that shows the subconcept relationships[4] between the concepts. The structure of the lattice is governed by the basic theorem for concept lattices:

$$\sup_{i \in I}(X_i, Y_i) = (\tau(\sigma(\bigcup_{i \in I} X_i)), \bigcap_{i \in I} Y_i)$$

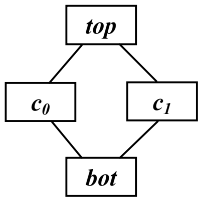
The theorem says that the least common superconcept (i.e. supremum denoted by *sup*) of a set of concepts can be computed by intersecting their intents, and by finding the common objects of the resulting intersection. Based on the basic theorem, a simple bottom-up algorithm for computing the concept lattice for a given context can be defined as follows [24]:

(1) Start with the bottom element of the concept lattice (*bot* in Figure 3), i.e. the concept consisting of objects that have all the attributes.

(2) Compute the *atomic concepts*, i.e. the smallest concepts with the extent containing each of the objects treated as a singleton set. In other words, consider each object $o \in O$ in turn, and identify the attributes of o. Those attributes become the intent of a potential new atomic concept c. The extent of c is formed from o and all other objects that have the attributes enumerated in the intent of c. Finally, accept c as an atomic concept if its extent is smaller than the extents of all those other concepts whose extents also contain o.

(3) Form a work list W containing all pairs of atomic concepts (c', c) such that c is not a subconcept of c' and vice versa. While W is not empty, remove a pair (c_a, c_b) from W. Then compute the supremum of c_a and c_b and assign it to c". If c" is a concept yet to be discovered then add all pairs of concepts (c", c) such that c is not a subconcept of c" and vice versa to W. The process is repeated until W is empty.

The analysis of the context given in Table 2 yields the concept lattice depicted in Figure 3. Besides the bottom and top concepts, the lattice contains two atomic concepts: c_0 that represents the base class (called *TestCase*) and c_1 that represents subclasses of *TestCase* overriding the *setUp* method.



top	({*TestCase*, *AccountTest*, *MoneyTest*}, ∅)
c_1	({*MoneyTest*, *AccountTest*}, {*inherits TestCase*, *overrides setUp*})
c_0	({*TestCase*}, {*name TestCase*})
bot	(∅, {*name TestCase*, *inherits TestCase*, *overrides setUp*})

Figure 3. A concept lattice of classes related to fixture setup

5.3 Relevancy of Program Elements

The example given in Table 2 and Figure 3 is unrealistic because the input for analysis was deliberately narrowed down to contain only those program elements that are relevant for setting up test fixtures in JUnit[5]. In reality, the input for the pattern extraction process can be as large as the whole implementation code of a framework and a set of example applications. It is clear that the input always contains lots of details that are not relevant to the current hot spot.

Without any further modifications the basic outline of the method given above produces one huge pattern that will contain all information in the input source code. In practice this is not a workable solution. To avoid being forced to apply concept analysis to excessively large contexts and to enable more precise interpretation of the results, a way to filter out irrelevant input before analysis must be introduced.

[4] A concept is a subconcept of another concept if its extent is a subset of the other concept's extent.

[5] We have omitted also the fixture classes and the internals of the methods to keep the example simple.

The solution is to select only those program elements and their properties that are relevant to the hot spot h at hand. No other elements and properties in the input will be considered. Here a *relevant property* is a property that corresponds to a property value $V \in \wp(E_{Java})$ for which a *relevancy function* $r_h(V) = 1$. In principle, r_h can have any suitable definition. In the following, we will simply assume that it is a mapping r_h: $\wp(E_{Java}) \rightarrow \{0, 1\}$ defined by giving a set of program elements $E_h = \{e_1, e_2, ..., e_k\} \subset E_{Java}$ so that $r_h(V) = 1$, if $V \cap E_h \neq \emptyset$, and $r_h(V) = 0$ otherwise. Using these definitions we can give the following general selection criteria:

(1) Select as objects only those program elements that have at least one relevant property.
(2) Select all relevant properties of all selected elements as attributes.
(3) If a property value V is a set consisting of multiple elements, then introduce an attribute for each relevant member of V (i.e. for each such $v \in V$ for which also $v \in E_h$).

This means that once a relevancy set E_h is identified it is possible to mechanically produce a relevant context (with respect to the hot spot h) for the concept analysis. Note, however, that it is up to the user to provide the relevancy information. The user can experiment with relevancy sets until she gets results whose scope and precision match her needs or she can define multiple relevancy sets to split the input into groups that can be handled separately.

For example, to get a context that describes the hot spot of setting up a test fixture for a test case in a JUnit application (recall Figure 1), we must identify those classes (and their internal properties) that are (or seem) relevant to that hot spot. To do so, we would add the roots of the relevant inheritance hierarchies (e.g. *TestCase* and *Money*) and other key elements (e.g. the *setUp* method in *TestCase*) into the relevancy set. Once that hot spot would be satisfactorily specified, we could move on to define relevancy sets for the other JUnit hot spots.

In practice, the required relevancy information implies a need for some initial knowledge of the structure and hot spots of the framework that is being analyzed[6]. So the main advantage of this approach is not so much in finding framework hot spots, but in being able to easily generate precise reuse specifications for those hot spots that are known or expected to exist in the implementation.

6. TRANSLATING CONCEPTS TO ROLES

In order to be able to translate the concepts resulting from an analysis of a context to the roles of a specialization pattern, we must define an unambiguous mapping from a set of concepts to a set of roles. The mapping will be based on the extents of the concepts. The objects belonging to the extent of a concept are precisely those program elements that are intended to be playing the role corresponding to the concept.

In the translation process we are looking for a set of roles where each program element (i.e. each concept analysis object) plays exactly one role (i.e. belongs to exactly one extent)[7]. Unfortunately, in a general case an object may belong to a number of extents in a concept lattice (for instance, *TestCase* belongs to the extents of c_0 and *top* in Figure 3). However, we can form a *concept partition* (i.e. a set of concepts where each object takes part in exactly one concept) from any concept lattice [25]. The concepts in the partition can then be translated to a set of roles that forms a model for that piece of code which the extents of the concepts were originally derived from. In Figure 3, concepts c_0 and c_1 form a partition that corresponds to the first two class roles (*TestCase* and *UserTestCase*) of the specialization pattern given in Figure 1.

6.1 Mapping Extent to Name and Cardinality

Once a concept partition that describes the relevant portion of the input at the right level of abstraction has been selected, we can translate the concepts in the partition into a specialization pattern. The translation starts with the creation of the roles of the pattern. A new role will be created for each concept in the selected partition.

The translation of a concept C into a role R_C is quite straightforward. The role kind is determined by the element kind of the objects included in the extent X_C of C. In other words, we create a class role if the concept describes classes, a method role for methods, and so on. The role's name and its location in the declaration hierarchy follow directly from the corresponding properties of the originating program elements. The declaration location for R_C will be under a role that stands for the declaring program elements of the elements in X_C. If the elements in X_C are top level elements (e.g. classes) R_C will be a top level role.

The cardinality of R_C is based on the extent X_C. By default the cardinality is exactly one. If there is at least one element corresponding to the declaring role R of R_C that does not have a child element in R_C then the cardinality will be from zero to one. This is because the input confirms that it is not mandatory for the elements corresponding to R to have a child element corresponding to R_C.

If there are multiple elements in X_C and R_C does not refer to any other role whose cardinality is from zero or one to infinity, then the cardinality is set to zero to infinity. This rule is based on the observation that, in general, having multiple elements in X_C implies that R_C is a role that describes potentially many implementation structures (e.g. various subclasses of a base class). On the other hand, if R_C refers to another role R_D (based on a concept D's extent X_D) that may have multiple elements associated with it, it is possible that the many elements in X_C can be interpreted to be in a "one-for-each" relationship with the elements in X_D. In such a case the correct cardinality is exactly one.

As an example of a concept, consider c_0 in Figure 3. It has one class (*TestCase*) in its extent. Therefore the result of translating c_0 is a top level class role (see Figure 1) with the same name as the name of the only class in c_0's extent. The size of the extent (one

[6] Although, in theory, one could start with a relevancy set that consisted of all program elements and continue by breaking the initial relevancy set into suitable subsets based on the initial results.

[7] It is usual for a program element to play multiple roles. However, those roles usually belong to separate patterns (or hot spots). In the extraction process we are interested in one pattern at a time so this restriction is not a serious one.

element) determines also the cardinality of the role to be exactly one. The other interesting concept in Figure 3, c_1, on the other hand, has two classes (*MoneyTest* and *AccountTest*) in its extent, so the cardinality for the corresponding role will be from zero to infinity. The name of the role can be derived from the names of the classes in c_1's extent, or as is the case here, the name of the common super class can be used as the basis for the role name.

6.2 Mapping Intent to Constraints

The constraints for each role R_C are defined based on the intent Y_C of the corresponding concept C. More precisely, for each attribute a in Y_C a *property lookup* is performed in R_C. If R_C can contain a constraint c_a corresponding to a then c_a will be declared for R_C. Both the type (e.g. an *inheritance* constraint) and value (e.g. (*class*, *Base*)) of c_a are determined by the property (e.g. (*inherits*, (*class*, *Base*))) corresponding to a. If the value is a reference to an element that corresponds to another role R_D (i.e. belongs to the extent of the associated concept D) then a reference to R_D is used as the value for c_a.

The property lookup may fail for some attributes. These attributes can just be ignored. Typically they would include those attributes that will be realized as roles on the next level in the declaration hierarchy. For example, an attribute (*overrides*, (*method*, *m*)) denoting that a class overrides a method *m* might not appear as a constraint in the class role, but it could imply that a method role will be declared within the class role later on. The attributes for which the property lookup fails are not redundant: they classify objects into separate concepts just as the other attributes.

Applying the abovementioned rules to c_1 in Figure 3 yields the inheritance constraint in *UserTestCase* that refers to *TestCase* (recall Figure 1). The other attribute in c_1's intent (*overrides setUp*) does not cause any further constraints to be added to *UserTestCase* because there is no overriding constraint available for class roles.

7. REVERSE ENGINEERING THE REUSE INTERFACE OF JUNIT

We have implemented a prototype tool (*Pattern Extractor*) that realizes the method and algorithms described in the previous sections [30]. The tool produces *Fred specialization patterns* from Java source code. *Fred* (*Framework Editor for Java*) is a programming environment intended for aiding framework-based software development [15, 16, 29]. It implements a specification language that can be used to model the hot spots of a framework. The language defines the hot spots in terms of role-based specialization patterns as described in Section 3.

Pattern Extractor produces specialization patterns based on the framework implementation associated with a representative set of example applications that are given to it as input. The user can then refine the automatically generated initial versions of the patterns to get a complete specification, which Fred can interpret to provide goal-oriented assistance for application developers using the framework.

Pattern Extractor implements the concept analysis based pattern extraction process described in Figure 2. The input and relevancy criteria provided by the user are used to produce a concept analysis context. The context is handed over to the concept analysis subsystem that implements the general concept analysis algorithms, such as building a concept lattice and calculating the concept partitions (recall Sections 4 and 5). Finally, the tool translates the selected concepts into roles and constraints as described in Section 6.

7.1 JUnit Framework

To test the effectiveness of the method introduced in this paper we have applied Pattern Extractor to automatically extract a Fred model for the JUnit framework [11]. JUnit is a framework for implementing systematic unit testing of Java programs. It consists of about 50 classes of which less than 20 belong to the core framework. The rest of the classes are UI components, extensions, and samples. JUnit was chosen as an example framework because it is commonly known, mature, simple, well designed, implemented in Java, and freely available.

Figure 4 shows a partial class diagram of JUnit (the *junit.framework* package) together with some application-level classes (the lower part of the diagram) that exemplify how the constructs of the framework can be extended and used. There are also the four most important hot spots of JUnit highlighted in the figure (see the legend in the lower right corner). The *DefiningTests* hot spot involves subclassing the *TestCase* class from the framework. The subclasses (e.g. *MoneyTest* and *AccountTest*) must implement the initialization of the *name* field in *TestCase*. This can be done most conveniently by defining a constructor that takes a name as an argument and then passes it over to the superclass via calling the constructor of the superclass. The test classes also define a number of test scripts (e.g. *testAdd* and *testDeposit*) that call methods of the classes to be tested (e.g. *add* in *Money* or *deposit* in *Account*). Each test script interacts with the objects to be tested and finally verifies the expected results with assertions.

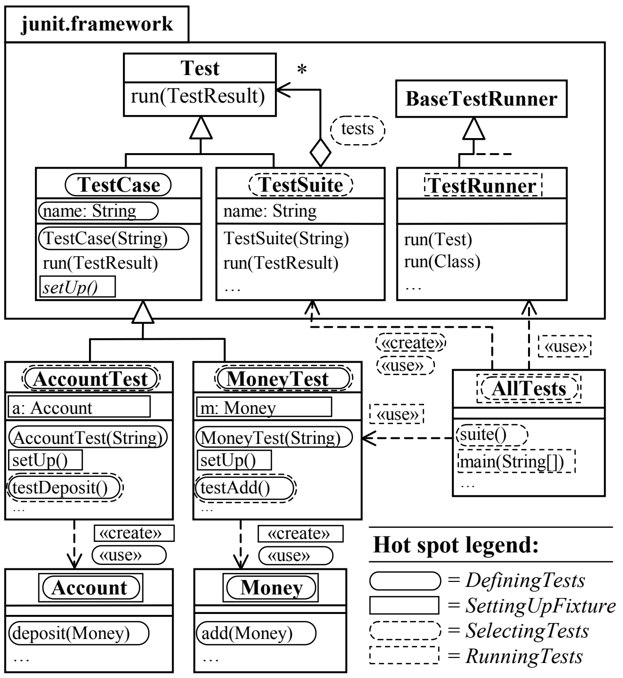


Figure 4. The main hot spots of the JUnit framework

The creation of the test material objects can be done in the test scripts themselves, but in order to be able to share these fixture definitions among scripts, the developer must use the *SettingUp-*

Fixture hot spot to define the initialization (e.g. *MoneyTest.setUp*) for fixture attributes (e.g. *MoneyTest.m*) by overriding the corresponding method declared in *TestCase*. Setting up a test fixture most often involves creating one or more instances of the classes to be tested (e.g. *Money*, *Account*).

The *TestSuite* class offers a possibility to combine test cases into tree hierarchies through its *tests* attribute. From the application developer's point of view this functionality is useful for selecting and grouping together those tests that should be run together. The *SelectingTests* hot spot allows the developer to do exactly that.

JUnit provides different test runners, which can run a test suite and collect the results. A test runner either expects a static *suite* method as the entry point to get a test to run or it will extract the suite automatically. *RunningTests*, the fourth hot spot depicted in Figure 4, is meant for selecting the tests to be run and an appropriate test runner class (e.g. *TestRunner*) to execute them. This can be accomplished by defining a *main* method (e.g. *AllTests.main*) where the runner's static *run* method is called with a test case class as an argument. All test scripts defined in the given test case will be looked up and executed through reflection.

7.2 Applying Pattern Extractor to JUnit

In a case study, eight specialization patterns were extracted to specify the reuse interface of JUnit (see Table 3). Four of them (*DefiningTests*, *SettingUpFixture*, *SelectingTests*, *RunningTests*) can be characterized as basic patterns, i.e. patterns that are involved in any specialization of JUnit. They correspond directly to the hot spots represented in Figure 4. The other four patterns describe hot spots that are relevant to more advanced or infrequent ways of using the framework.

The input given for the extraction process consisted of the source code packages of the core JUnit framework package (*junit.framework*) and its extensions (*junit.extensions*, *junit.runner*, and *junit.textui*) together with a set of sample tests (*junit.samples.money*)[8].

Table 3 summarizes the results of the case study. It shows the characteristics of each produced specialization pattern including the name, the number of non-empty lines in the extracted textual pattern specification (a measure for the pattern size), and the relevancy set[9] used in the extraction.

The automatically extracted patterns were adjusted manually to achieve the overall quality and usability level of the hand-written specifications made for various frameworks within the Fred project (see, e.g., [29]). The last two columns in Table 3 give an estimate of the quality of the extraction process. The *strict extraction percentage* shows the portion of the specification that consisted of those automatically produced lines that could be used directly without any adjustments. The *loose extraction* percentage, on the other hand, counts in also those specification lines that only needed trivial modifications (e.g. updating descriptions to better suit the context). It also omits the removed lines. The strict and loose extraction percentages define thus a range in which the effective net benefit of this approach probably lies.

[8] The *AccountTest* class was added to the money package to complement the *MoneyTest* class provided with the distribution.

[9] An asterisk is used as a short hand notation for all program elements whose name begin with the given string.

Table 3. Characteristics of the extracted JUnit patterns

Pattern	*Lines*	*Relevancy set*	*Strict %*	*Loose%*
DefiningTests	114	{*TestCase*, *TestCase.TestCase*, *MoneyTest.test**, *MoneyTest.MoneyTest*, *Money*}	22	58
SettingUp-Fixture	63	{*TestCase*, *TestCase.setUp*, *TestCase.tearDown*, *Money*}	57	93
SelectingTests	122	{*TestCase*, *TestSuite*, *MoneyTest.test**, *AccountTest*, *AccountTest.test**, *AllTests*, *AllTests.suite*}	21	65
RunningTests	54	{*BaseTestRunner*, *TestRunner*, *TestRunner.run*, *AllTests*, *AllTests.main*}	37	85
Decorating-Tests	103	{*TestDecorator*, *TestDecorator.TestDecorator*, *TestDecorator.run*, *AccountTest*, *AccountTest.testDelayedTestCase**, *TestCase*}	45	71
Running-RepeatedTests	47	{*RepeatedTest*, *AccountTest*, *AccountTest.testMultipleTimes**, *TestCase*}	49	82
RunningTests-InThreads	61	{*ActiveTestSuite*, *AccountTest*, *AccountTest.testInThreads**, *TestCase*}	30	74
Testing-Exceptions	69	{*Exception*, *ExceptionTestCase*, *AccountTest*, *AccountTest.testException**, *AccountExceptionTestCase*}	32	67
Total	633		-	-
Average	79		37	74

To give an idea of what the generated specialization patterns looked like and how they were adjusted, we provide a listing of a JUnit specialization pattern (see *SettingUpFixture* below)[10]. Note that the parts of the source text that were removed from the final pattern are written in italics, and the parts that were manually added or changed are written in bold.

```
pattern SettingUpFixture;
class TestCase {
  description "Click <a href="…">here</a> …";
  method setUp {
    description "Click …";
  }
}
class UserTestCase* {
  inherits TestCase;
  description "Click …";
  method setUp {
    overrides TestCase.setUp;
    description "Click …";
    fragment fixtureCreation {
      description "This code snippet …";
      source "<#fixtureAttr.name> =
        new <#fixtureAttr.type>(12, "CHF");
        <#…> = new <#…>(14, "CHF");"
    }
  }
  field fixtureAttr* {
    isTypeOf FixtureClass;
    description "All fixture objects …";
  }
}
class role FixtureClass* {
  description "A fixture class is a class …";
}
```

[10] The listing provides a slightly compacted version of the pattern (e.g. some of the descriptions have been shortened and the *tearDown* method role has been omitted); hence the difference between the number of lines in the listing and in Table 3.

7.3 Evaluation of the Results

The results of our case study indicate that about half of the specialization pattern code for modeling the reuse interface of a framework can be automatically extracted from source code with our method. Of the other half that needs to be manually given or at least modified to suit the context, most consists of code fragment roles. This results from the fact that analyzing patterns from method bodies (expressions and statements) is a hard task in general. In addition, Fred does not currently support method bodies in its AST representation of the source code. That is why the extraction of the code fragment roles is based directly on the tokenized source text, which allows only very primitive analyses.

Our results show that automatic extraction of specialization patterns yields quite an accurate overall picture of the structures of the patterns, and that it is indeed only the details that need further modifications to make the patterns usable. In addition to the code fragment roles, majority of the additions and modifications were related to informal textual templates associated with roles (such as descriptions and default names) rather than to actual constraints.

Based on our experiences with Fred and Pattern Extractor, we can conclude that it is possible to describe the intended rules governing the framework's specializations with a precise role-based formalism. It is clear that a thorough specification of a framework's reuse interface raises the development costs. However, we argue that these costs are relatively low when compared to the development costs on the whole and, furthermore, the savings gained in training and mentoring will be substantial, especially if many users are going to use the same framework.

It has already been mentioned that the effectiveness of our reverse engineering approach depends on the number and quality of the examples that are available as input. This approach is not meant to replace the usual process of getting the specification of reuse interfaces from the developers early in the design stage. It is clear that specifying the reuse interface of a framework as early as possible is beneficial for the framework development process and ultimately also for the framework users. The motivation for this work has been to assist framework users (and developers) in situations where the documentation provided with the framework is not detailed enough to directly enable tool support.

Using the presented approach, it is possible to automatically generate a considerable portion of a reuse interface specification. This makes reuse interface modeling faster and allows even modeling of frameworks that are still under development. Automation can also increase the quality of the produced models, because it may reveal shortcomings, omissions, or inconsistencies in manually prepared reuse models made for the same framework.

A major obstacle in design recovery is the scalability of the algorithms being used. The worst-case running time of the concept lattice construction algorithm used in this work is exponential [27]. In practice, however, the running time is polynomial and thus feasible even for rather large contexts because there typically are $O(n)$ or $O(n^2)$ concepts in a lattice with n objects as opposed to the maximal $O(2^n)$ concepts.

Tonella and Antoniol use concept analysis to recover (design) patterns from source code [28]. They define all possible *combinations* of classes as objects, which forces them to limit their analyses to combinations with fewer than five classes when using any non-trivial system as input. In our approach we define one program element (a class, a method, and so on) as an object. This means that that the size of the patterns that can be extracted and the size of the systems used as input are not as limited.

8. CONCLUSION AND FUTURE WORK

We have introduced a new approach for specifying framework reuse interfaces in order to facilitate tool-supported framework specialization. The method presented in this paper automates many trivial details of the modeling process by accurately extracting most roles and constraints from the framework source code and existing example applications.

Structuring a software system reflects design decisions that are inherently subjective [25]. Similarly, there is always a creative element in preparing documentation or a specification for a reusable system. This means that substantial user interaction will be required in any approach that tries to assist, e.g., modeling a framework reuse interface. In the Pattern Extractor tool introduced in this paper, the user is responsible for structuring the model into separate specialization patterns by specifying appropriate relevancy criteria. The tool selects relevant program elements according to the criteria and uses the selected elements as input for the concept analysis. The analysis, in turn, produces a concept lattice that is automatically translated into a specialization pattern.

Even though it is unlikely that framework reuse interface specification can be fully automated, there is still a lot of room for improvements in Pattern Extractor. Further research is needed to build a precise model of how the selection of input and relevancy criteria affect the produced specialization patterns. Based on a detailed model it would be easier to make justified decisions on which properties of the program elements should be used as concept analysis attributes and how the input should be divided into separate contexts to produce structured models.

The reverse engineering method described in this paper must be applied to a variety of frameworks in order to truly validate its potential. As our experiences accumulate we expect to be able to formulate practical and generally applicable guidelines for defining the relevancy criteria to easily produce useful specialization patterns. At this point, however, it must be admitted that we cannot present a full assessment of our method with respect to the risk of getting irrelevant results depending on the input selection.

In our view, finding the main concepts of a framework from its documentation is the key to understand the framework's reuse interface. At the same time it is also an essential precondition of using Pattern Extractor successfully. If no documentation is available the same basic information must be gathered from the system's source code. Systematic analysis of the framework's class hierarchy as described in [29] and automatic pattern detection (see, e.g., [20]) are good ways to get started. Still, there clearly is a need for further research on automatic program analysis methods, especially for methods that concentrate on analysis of reusable assets, such as frameworks.

9. REFERENCES

[1] Basili V., Briand L., Melo W., How Reuse Influences Productivity in Object-Oriented Systems. *Communications of the ACM* 39, 10, 1996, 104-116.

[2] Booch G., Object Solutions: Managing the Object-Oriented Project. Addison-Wesley, 1996.

[3] Codenie W., De Hondt K., Steyaert P., Vercammen A., From Custom Applications to Domain-Specific Frameworks. *Communications of the ACM* 40, 10, 1997, 71-77.

[4] Chikofsky E., Cross II J., Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* 7, 1, 1996, 13-17.

[5] Deutsch L., Design Reuse and Frameworks in the Smalltalk-80 System. In: Biggerstaff T., Perlis A. (eds.), *Software Reusability Vol. II*, ACM Press, 1989, 57-71.

[6] Durham A., Johnson R., A Framework for Run-time Systems and Its Visual Programming Language. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, San Jose, California, USA, October 1996, ACM SIGPLAN Notices 31, 10, 1996, 406-420.

[7] Eisenbarth T., Koschke R., Simon D., Incremental Location of Combined Features for Large-Scale Programs. In: Proc. *International Conference on Software Maintenance (ICSM 2002)*, Montreal, Canada, October 2002, IEEE Computer Society Press, 273-283.

[8] Fayad M., Schmidt D., Object-Oriented Application Frameworks. *Communications of the ACM* 40, 10, 1997, 32-38.

[9] Froehlich G., Hoover H., Liu L., Sorenson P., Hooking into Object-Oriented Application Frameworks. In: Proc. *19th International Conference on Software Engineering (ICSE'97)*, Boston, Massachusetts, USA, May 1997, IEEE Computer Society Press, 491-501.

[10] Fayad M., Schmidt D., Johnson R., (eds.), *Building Application Frameworks — Object-Oriented Foundations of Framework Design*. Wiley, 1999.

[11] Gamma E., Beck K., JUnit: A Cook's Tour. *Java Report* 4, 5, 1999, 27-38.

[12] Ganter, B., Wille, R., *Formal Concept Analysis — Mathematical Foundations*. Springer, 1999.

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

[14] van Gurp J., Bosch J., Design, Implementation, and Evolution of Object Oriented Frameworks: Concepts and Guidelines. *Software — Practice & Experience* 31, 3, 2001, 277-300.

[15] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Generating Application Development Environments for Java Frameworks. In: Proc. *3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01)*, Erfurt, Germany, September 2001, Springer LNCS 2186, 163-176.

[16] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Annotating Reusable Software Architectures with Specialization Patterns. In: Proc. *Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, Amsterdam, Netherlands, August 2001, IEEE Computer Society Press, 171-180.

[17] Johnson R., Documenting Frameworks Using Patterns. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'92)*, Vancouver, British Columbia, Canada, October 1992, ACM SIGPLAN Notices 27, 10, 1992, 63-76.

[18] Krasner G., Pope S., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1, 3, 1988, 26-49.

[19] Ortigosa A., Campo M., Salomon R., Towards Agent-Oriented Assistance for Framework Instantiation. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, Minneapolis, Minnesota, USA, October 2000, ACM SIGPLAN Notices 35, 10, 2000, 253-263.

[20] Paakki J., Karhinen A., Gustafsson J., Nenonen L., Verkamo I., Software Metrics by Architectural Pattern Mining. In: Proc. *International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, Beijing, China, August 2000, 325-332.

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

[22] Quilici A., Reverse Engineering of Legacy Systems: A Path Toward Success. In: Proc. *17th International Conference on Software Engineering (ICSE'95)*, Seattle, Washington, USA, April 1995, IEEE Computer Society Press, 333-336.

[23] Rine D., Nada N., Three Empirical Studies of a Software Reuse Reference Model. *Software — Practice & Experience* 30, 6, 2000, 685-722.

[24] Siff M., Reps T., Identifying Modules via Concept Analysis. In: Proc. *International Conference on Software Maintenance (ICSM'97)*, Bari, Italy, October 1997, IEEE Computer Society Press, 170-178.

[25] Siff M., Reps T., Identifying Modules via Concept Analysis. TR-1337, Computer Sciences Department, University of Wisconsin, Madison, WI, 1998.

[26] Shull F., Lanubile F., Basili V., Investigating Reading Techniques for Object-Oriented Framework Learning. *IEEE Transactions on Software Engineering* 26, 11, 2000, 1101-1118.

[27] Snelting G., Reengineering of Configurations Based on Mathematical Concept Analysis. *ACM Transactions on Software Engineering and Methodology* 5, 2, 1996, 146-189.

[28] Tonella P., Antoniol G., Object Oriented Design Pattern Inference. In: Proc. *International Conference on Software Maintenance (ICSM'99)*, Oxford, England, August-September 1999, IEEE Computer Society Press, 230-239.

[29] Viljamaa A., Pattern-Based Framework Annotation and Adaptation — A Systematic Approach. Licentiate thesis, Report C-2001-52, Department of Computer Science, University of Helsinki, 2001.

[30] Viljamaa J., Automatic Extraction of Framework Specialization Patterns. Licentiate thesis, Report C-2002-47, Department of Computer Science, University of Helsinki, 2002.

[31] Waters R., Chikofsky E. (eds.), Special Section on Reverse Engineering. *Communications of the ACM* 37, 5, 1994, 22-93.