Documenting Maintenance Tasks Using Maintenance Patterns

Imed Hammouda
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
imed@cs.tut.fi

Maarit Harsu
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
harsu@cs.tut.fi

Abstract

A common problem in software maintenance is the lack of documentation required for carrying out the maintenance tasks. Both expected and unexpected maintenance tasks use and produce important information, which is not systematically recorded during the evolution of a system. As a result, maintenance becomes unnecessarily hard. In this paper, we suggest the use of maintenance patterns to record information about maintenance tasks. Maintenance patterns are organized collections of software elements relevant for a particular maintenance concern. Such a pattern can be exploited for advanced tool support where the tool keeps track and guides the system developer step by step through a maintenance procedure.

1 Introduction

A truly successful software system is one that is continually modified and enhanced [22]: prolonging the lifespan of a software system has always been one of the main purposes of software maintenance. However, a common problem in software maintenance is the lack of documentation needed for the maintenance tasks. In fact, it is well known that maintenance should be considered already in the design phase [21]. Unfortunately, this suggestion is usually not followed, partly due to the limitations of the current technologies and methodologies. Even though a designer anticipates certain maintenance tasks during design, there are no systematic means to record this information in such a form that it could be used to provide tool-assisted guidance for the maintainers. In the case of an unanticipated maintenance activity, the information concerning the activity is typically lost completely, or documented scarcely at best.

However, this information would be valuable in the future maintenance, because it reveals a logical connection between different parts of the system that is relevant from the maintenance viewpoint. Such information is valuable if similar maintenance tasks are repeated, or if other maintenance tasks concern parts that have been involved in previous tasks. In the latter case, information about previous maintenance tasks can be exploited to guarantee that the new maintenance task does not corrupt the effect of the previous action. Again, this information can be used to provide tool support for repeating similar maintenance tasks, for reversing maintenance tasks, and for sustaining compliance with earlier maintenance tasks.

In this paper, we introduce a way to document software maintenance via so-called *maintenance patterns*. In this context, a pattern means a specification of a set of software entities that are logically related from the viewpoint of some concern. More precisely, a pattern consists of roles that can be bound to program entities (like classes, methods and attributes) and a set of constraints that these bindings must satisfy. A pattern can exist without any bindings, or it can be partially or fully bound. A maintenance pattern is a pattern that captures the set of program elements involved in a logical maintenance task. The roles are used to store the information about the participants of a maintenance task whereas the constraints state the relationship that must be followed by the program elements bound to different roles. In addition to the constraints specifying the relationships of the program elements, a maintenance pattern can be associated with other information concerning the maintenance task, like informal instructions for the maintainer or default code templates for some program elements.

Maintenance tasks can be divided into smaller subtasks that require modifications to individual small program elements scattered around the system to be maintained. This gives maintenance patterns a cross-cutting nature. In this sense a pattern is conceptually close to an aspect [2], but since we will have a fairly specific technical realization of this concept, we want to avoid using this term. We will refer to the cross-cutting concern represented by a maintenance pattern as *maintainability concern*.

Software maintenance may include different tasks: the

system can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [1]. In fact, it is acknowledged that the maintenance phase in software development is the most difficult and thus causes major costs [5]. We think that these deficiencies can be partially solved by associating the system with maintenance patterns. In the paper, we show the benefits of applying such patterns in different maintenance situations such as system adaptation. We also show how errors resulting from maintenance tasks could be reduced when using our approach.

An implication of our approach of documenting maintenance tasks as patterns is that we can exploit generic pattern-based tool support for maintenance. In particular, we use a tool called JavaFrames (previously Fred) to implement the idea of maintenance patterns. JavaFrames [12] is an Eclipse [3] plug-in which allows the specification and use of patterns in a Java programming environment. Given a pattern specification, the tool provides task-driven guidance for a developer to carry out the binding of the roles as required by the pattern. Although the tool was originally developed to support the specialization of a framework, we argue that the generic pattern concept of JavaFrames can be used as a basis of maintenance patterns as well. The benefit of exploiting JavaFrames is that various kinds of useful tool support become readily available:

- A task list can be generated automatically on the basis of a maintenance pattern, to guide the maintainer in carrying out the maintenance task foreseen by a designer (and expressed as a maintenance pattern).
- If a performed maintenance task is recorded as a maintenance pattern, the tool keeps track of the different program elements involved in the task and enforces the relationships between those elements, as specified by the constraints in the pattern.
- If some program element involved in a maintenance action can be deduced on the basis of the other elements involved in that task, the former element can be automatically produced.
- Various kinds of textual information can be conveniently associated with single maintenance actions. Such information can be used to guide the maintainer to carry out the maintenance action, or to store explanations of the maintenance action in a form that can be easily retrieved during the evolution of the system.

The rest of the paper proceeds as follows. In Section 2 we discuss maintenance patterns and their use in more detail. We give three concrete examples of well known maintenance problems and show how these problems can be solved using maintenance patterns. In Section 3 we discuss the JavaFrames tool which serves as our basis of the realization of maintenance patterns and show how to use the tool in applying maintenance patterns. Section 4 discusses maintainability concern architectures in general, while Section 5 considers maintainability concerns of JavaFrames itself. In Section 6 we present related work and finally conclude in Section 7.

2 Using maintenance patterns

In this section we first discuss the conceptual basis of maintenance patterns. Then we introduce three examples that apply maintenance patterns.

2.1 Maintenance patterns

A maintenance pattern consists of roles, associated with constraints and properties. A role has a type which determines the kind of program elements that can be bound to a role. For example, a class role can be bound to a concrete Java class. We assume that there are appropriate role types covering the basic Java concepts. In addition, a note role can be bound to an acknowledgment provided by the maintainer. A note role is used to associate informal instructions at certain points in the maintenance pattern.

Constraints are structural conditions that must be satisfied by the program element bound to a role. For example, a constraint of class role *P* may require that the class bound to *P* must inherit the class bound to another role *Q*, or that the method bound to role *R* must be located within the class bound to role *S*. The properties include various additional specifications that are useful in practice, like a short (informal) description of the binding task or a default implementation for the program element to be bound (used for automatically generating the program element). A property can also define the multiplicity of a role, that is, the lower and upper limits for the number of the instances of the role. For example, if a method role has multiplicity 0..1, the method is optional in the pattern, because the lower limit is 0.

We assume tool support which can generate a task list for a partially bound maintenance pattern. The task list displays a task for each role that can be instantiated and bound in the present situation, taking into account the dependencies (implied for example by the constraints) and the multiplicities of the roles. We also assume that the tool is integrated with a program editor and that the constraints are checked after every editing action. We will not explain in detail how the tool does all this; an account of the mechanisms is given in [12]. Every time a binding action is performed, the tool generates an updated task list (because new tasks will probably become doable). Thus, each binding action represents one maintenance action within the maintenance task managed by the maintenance pattern, and the task list generated

by the tool provides a step-by-step instruction on how to carry out the maintenance task. Usually a binding action means that the maintainer should provide a new program element at a certain place, modify an existing program element, or simply check manually that the code satisfies certain requirements. The latter can be accomplished using note roles.

With this rough idea of a maintenance pattern in mind, in the next subsections we give three example situations where a maintenance pattern is applied. We will give a more detailed description of the maintenance patterns in terms of JavaFrames in Section 3.

2.2 Extending an existing system

As an example of adaptive maintenance, consider a typical situation in an object-oriented system where the system should be modified by providing a new implementation for a certain concept represented by a base class or an interface. This kind of maintenance task is often anticipated, and indeed maintenance may have been the driving concern in the design solution leading to the use of a base class or an interface.

The above situation is depicted in Figure 1. Originally, we have three classes: a base class *Concept*, a second class *ConcreteImplA* extending *Concept*, and another arbitrary class *Client* that instantiates *ConcreteImplA*. At some point, the maintainer of the system finds herself in a situation where she should provide a new specialization of *Concept* that replaces *ConcreteImplA* with *ConcreteImplB*. If the system is well-designed, references to *ConcreteImplA* (like instantiations) are isolated to few places, but these places may nevertheless be hard to locate. The designer should therefore document the places as a maintenance pattern (called, say, "New implementation of *Concept*"). Let us assume here that the only place where *ConcreteImplA* is referenced is the instantiate method of *Client*.

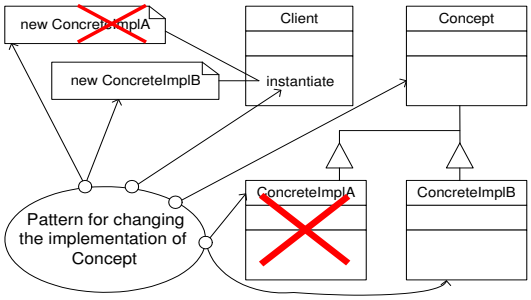

Figure 1. Extending an existing system.

Figure 1 uses a modified version of UML notation for patterns because our pattern concept differs slightly from that of UML. The patterns are denoted by ellipses. The tiny circles on the boundary of the patterns denote the roles that are associated with a maintenance pattern. For instance, the role bound to the superclass *Concept* suggests that there is a single code point (class *Concept*) played by the pattern role. The multiplicity of a role can be more than 1. The role referring to both *ConcreteImplA* and *ConcreteImplB* is an example of a role that refers to more than one code point playing the same role.

When applying the maintenance pattern in the situation described above, all required modifications are presented as simple maintenance tasks. The pattern has roles referencing the class *Concept* and the code points where its implementation class is instantiated. Another role is used to refer to the implementation class itself. When the user decides to change the existing implementation class by a new one, the pattern enforces the inheritance relationship between *Concept* and the new implementation class. In addition, the pattern warns the user to revisit the code points where the implementation class is used. For example, the pattern might ask the user to change the reference to *ConcreteImplA* in the method *instantiate* of class *Client* to a reference to *ConcreteImplB* instead.

2.3 Copy-paste programming

Newly-implemented programs are often based on earlier constructed ones. In this case, the programmer typically copies the older program and modifies it to suit the new purpose. In other situations, a new program may be constructed out of combining together several pieces of code extracted from a number of other programs. In both cases, the practice is to first copy and paste different pieces of code and then try to refactor the assembled elements. This coding technique is commonly used when maintaining existing systems as well, especially when the purpose of the maintainer is to adapt the system to a new environment or to improve some of the system qualities.

Copy-paste programming may add uncontrolled errors to the system. This is illustrated in the case where the original piece of code, i.e. the code which has been copied, has been modified in order to correct some errors. Most probably, all code replica need to be modified, too. However, it is not easy to know where all the copied pieces lie. Finding out these code pieces can be a painful and costly activity.

Copy-paste programming is identified as an AntiPattern called cut-and-paste programming [6]. It is considered as a bad programming style, and the best choice is to avoid it. However, copy-paste style of program development is sometimes unavoidable, because refactoring the code to eliminate all code duplications would lead to complicated and unnatural decomposition of the system. Thus, copy-paste occurrences are a kind of micro aspects of the system: they represent an implementation level cross-cutting

concern that should be made explicit in the system. From a maintenance viewpoint, the above practice can be seen as lacking control over the maintenance activity. There is no clear documentation on what and where modifications should be carried out. Maintenance patterns can be used to document and keep track of copied program elements.

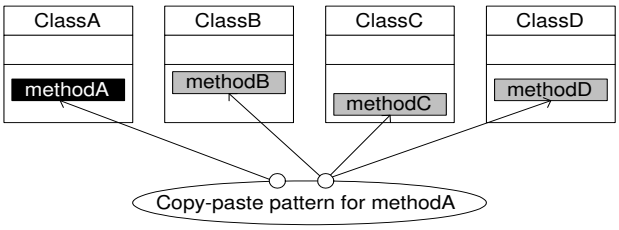

Figure 2. Pattern for copy-paste programming.

Figure 2 shows a maintenance pattern that manages the copy-paste style of programming. The pattern adds more control to the maintenance activity in the sense that the copied code (marked with dark color), is replicated and inserted in the right code points (marked with grey color) as specified by the pattern. The pattern has a role referring to the copied code and a second role referring to the code points where it has been pasted. When the copy-paste situation is identified, the programmer records this as copy-paste maintenance pattern, binding the roles to the concrete program elements. If the user modifies the original code, the pattern warns her to consider the replica by revealing their location. In some situations, a modified version of the copied piece is needed. In this case, too, the maintenance pattern contains the tracking information about the code pieces that are involved in the maintenance task.

Copy-paste programming is quite similar to implicit coding rules [16]. Implicit coding rules cannot be typically found in software documentations, instead, they are in the heads of experienced maintainers. An example of such a rule is that a certain global variable should be initialized before calling a certain routine. Violations of these rules are called bug code patterns. In the similar way, to handle copy-paste programming without violating the code the maintainer should know the occurrences of the copied code. Usually, such knowledge is not presented explicitly in software documentation.

2.4 Fragile base class problem

Let us assume a simple class inheritance situation, a derived class inherits from a base class. The fragile base class problem is the situation where modifications in the base class may cause the derived class to malfunction. The base class is said to be fragile because changes in it break its role as a base class. A thorough study of the problem is presented in [17]. Figure 3 shows an example of the fragile base class problem. A class library provides a class *Container* for storing objects. The class has two methods *add* inserting a new object in the container and *addSet* invoking the *add* method to add a set of objects to the container. Suppose that an application developer decides to extend *Container*. She derives a class *CountedContainer* introducing an instance variable *counter* and redefining the add *operation* to increment *counter* every time a new object is added to the container. At a later time, the library maintainer decides to optimize the class *Container*. She provides a new implementation for *addSet* without invoking *add* thinking that the new system is compatible with the previous one. Unfortunately, when the new version of *Container* is used as a base class for *CountedContainer*, the system returns the wrong number of objects since *counter* is not updated anymore.

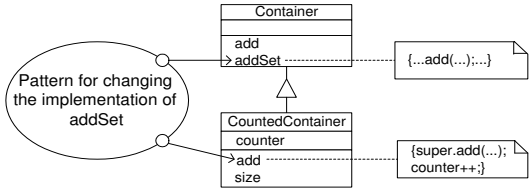

Figure 3. Fragile base class problem.

In this kind of situation, a maintenance pattern can be used to monitor changes in the superclass. The pattern has a role referring to the base class method and a second role referring to the subclass method being used by the superclass. Initially, the roles are bound to the concrete methods. If the maintainer modifies a referred method in the superclass, the pattern asks her to verify the related method in the derived class. The warning may take the form of an informal task description. Thus, possible problems related to the fragile base class case can be eliminated.

3 Maintenance patterns in JavaFrames

In this section, we first discuss the main features of JavaFrames tool. We then show how to present the maintenance patterns discussed in Section 2 and how to apply them using the environment.

3.1 JavaFrames environment

JavaFrames is initially a pattern-oriented task-based prototype tool for framework specialization. The tool has been used as a Java development environment and is currently

being adapted to support UML semantics [13]. However, the tool can be easily extended to support any other type of semantics.

The tool comes with several useful features. Three of these main qualities are:

Simple task lists : JavaFrames environment models a set of specification in the form of role-based patterns. When applying these patterns, the tool generates a number of tasks to be carried out in order to meet the specifications. By task, we mean a simple action that adds an element or enforces a property on the model. Tasks are kept simple enough so that their immediate result is easy to track and can be changed if needed. An example task is to provide a Java method.

Adaptive documentation : JavaFrames environment is able to record the history of the user's tasks and informs her what to do next using the recorded information. The tool, for example, uses the concrete names of the Java classes specified earlier by the user when documenting next related tasks. The concrete names can be used both in the informal task descriptions and the generated default code.

Integrated source editor : The integrated Java editor keeps track of the application of the patterns and checks the constraints during editing on the fly: constraints are re-checked every time the source is changed and the user is asked to repair the violations.

The tool implementation of the general pattern concept in addition to the qualities mentioned above makes it possible to adapt the environment to support software maintenance activities. For this purpose, we treat JavaFrames patterns as maintenance patterns. A maintenance pattern then consists of several roles, each role corresponds to a code point we are interested in during the maintenance activity. Role properties define how the maintenance task should be performed and the inter-role dependencies show when these maintenance tasks should be carried out. Violated constraints are also considered as maintenance tasks.

In JavaFrames, patterns are usually put in a hierarchical structure defining a unified context. During a maintenance session, the pattern roles are either bound to new program elements, for example by creating a new Java class for a class role, or bound to existing program elements. The action of associating pattern roles with concrete program elements represents a maintenance task the user should perform. When a task is performed, new maintenance tasks might be generated.

3.2 Presenting maintenance patterns in JavaFrames

In this subsection, we discuss the JavaFrames specification of the Extending Pattern shown in Figure 1. This maintenance pattern consists of four roles: two class roles, a method role, and a code snippet role. The class roles represent the base class *Concept* and the class that inherits from it. The method role specifies the method *instantiate* in class *Client*. Finally, the code snippet role stands for the implementation of the method *instantiate*.

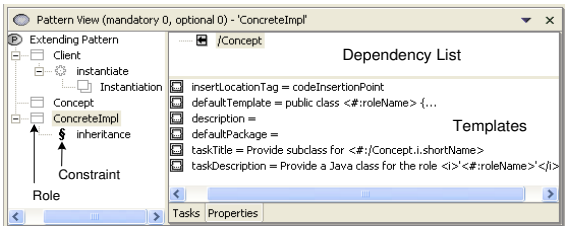

Figure 4. JavaFrames maintenance pattern.

Maintenance pattern: changing subclass	
Roles	**Properties**
Concept : class	**description**: system super class.
ConcreteImpl +: class	**taskTitle**: Provide subclass for <#:/Concept.i.shortName> **taskDescription**: Provide a Java class for the role '<#:roleName>'. The class should provid concrete implementation for the inherited class <#:/Concept.i.shortName>.
inheritance : constraint	**value**: <#:/Concept.i>
instantiate : method	**description**: method in client class, should instantiate <#:/ConcreteImpl.i.shortName>.
Instantiation : code snippet	**taskTitle**: Remember to instantiate class <#:/ConcreteImpl.i.shortName>. **defaultImplementation**: <#:/ConcreteImpl.i.shortName> clazz = new <#:/ConcreteImpl.i.shortName>();

Figure 5. Textual representation of the pattern.

Figure 4 shows the representation of the pattern in JavaFrames pattern editor. The left view displays the pattern roles and their constraints. The right upper view shows the dependencies of the selected role whereas the lower part exposes the properties of the role expressed as text templates. In order to clarify the structure of the pattern, we give in Figure 5 a more readable textual specification. Every role has a set of properties; the *taskTitle* property of the *ConcreteImpl* role for example, tells the maintainer what task should be performed. The definition of this property refers to *Concept* role in the form of `<#:/Concept.i.shortName>`, this tag refers to the concrete name of the superclass. The *inheritance* constraint

has value `<#:/Concept.i>` saying that any instance of the *ConcreteImpl* role (class *ConcreteImplA* and class *ConcreteImplB*) should inherit from instances of *Concept* role. The multiplicity symbol "+" that comes with *ConcreteImpl* means that there can be more than one program element playing the role *ConcreteImpl*. When no multiplicity is given, the role should be bound to exactly one program element, this is the case of the *Concept* role. Finally, the code snippet role *Instantiation* has default implementation that can be used by the maintainer resulting in a concrete action.

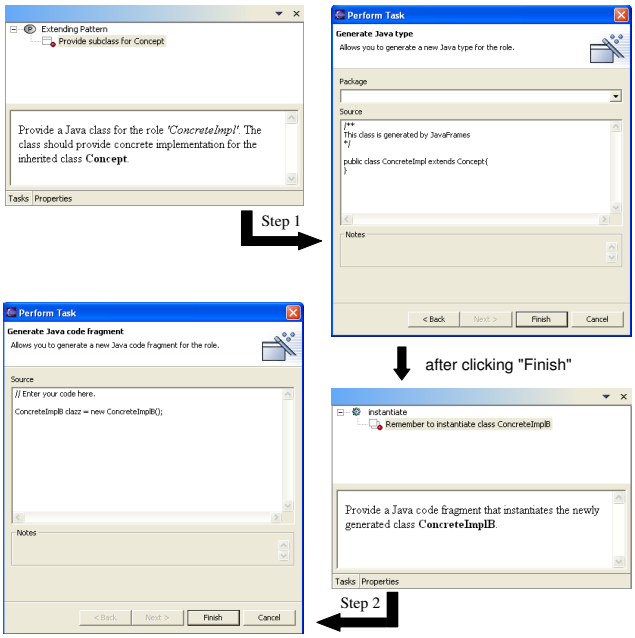

Figure 6. Sample maintenance tasks.

Figure 6 shows sample maintenance tasks to be performed when applying the maintenance pattern discussed above. In task 1, the user provides a new subclass *ConcreteImplB* inheriting from class *Concept*. The next task is to remind the user that the newly generated subclass has to be instantiated instead of the existing subclass. The environment generates default implementation and suggests the right class to be instantiated.

The other two maintenance problems mentioned in Section 2 can be handled with JavaFrames, assuming that the environment can detect changes in the source code. In order to achieve that, a new "unchanged source" constraint that is invalidated by any change in a bound element is needed. Support for such a constraint can be added to JavaFrames.

The pattern that manages copy-paste programming has two roles; a first role with a reference to the code to be copied and a second role with a reference to the program elements where the copied code should be pasted. There is a dependency from the latter role to the former role and there is an "unchanged source" constraint on the first role. When applying the pattern, the first task is to specify the original code (*methodA* in the example) whereas the remaining tasks are to provide the code points where to find the copied code (*methodB*, *methodC*, and *methodD* in the example). At a later time, when the maintainer decides to make changes in the original code, the "unchanged source" constraint is violated and the pattern warns the user to revisit the replica.

As for the fragile base class problem, the pattern has two method roles in addition to two class roles associated with the base class and the subclass. The first method role refers to the method of the base class (*addSet* in the example) whereas the second method role points to the method of the subclass (*add* in the example). There is a dependency from the first role to the second role since *addSet* invokes *add*. In addition, there is an "unchanged source" constraint on the first role. If the library maintainer performs changes in the base class method, the "unchanged source" constraint is violated and a task is generated warning her of possible problem in the subclass. The environment can suggest the user to add a copy of the base class method with the original implementation to the subclass.

From the discussion above, we see how JavaFrames environment can manage maintenance activities via simple task lists. The tasks are easy to follow and their actions can be visually controlled. The patterns are used to refer to the code points involved in the maintenance activity. In the task descriptions, concrete names such as *ConcreteImplB* in task 2 of Figure 6, are used. Moreover, default implementation required for the maintenance actions can be suggested. This offers the maintainer a stepwise adaptive maintenance environment. However, the number of maintenance tasks should be carefully planned since a huge number of tasks can annoy the maintainer and slow down the maintenance phase. A related problem is how to manage the maintenance activity of a large system via small-grained maintenance patterns instead of using one huge macro pattern. In the next section, we propose a possible solution to such problems.

4 Maintainability concern architectures

In large real systems maintenance patterns can be overlapping and co-operate with each other as illustrated in Figure 7. Several maintenance patterns are connected to each other via the accompanying classes which require maintenance actions. Each time a maintenance action (described with a maintenance pattern) occurs, the connected maintenance patterns should be checked to verify whether these patterns produce any modification requirements. Thus, maintenance patterns support change impact analysis to find the points of the system where the maintenance task extends.

In more detail, Figure 7 shows the code points that

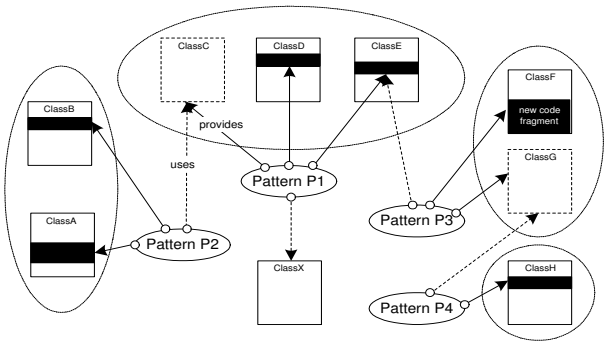

Figure 7. Maintenance patterns.

should be considered when maintaining a specific software system. Code points that are related to each other are grouped together under the same unit. This is marked in the figure by a circular shape. For example, the required modifications in *ClassA* and *ClassB* are tied together. The role of the maintenance pattern is to contain related code points. For instance, pattern *P2* groups the code points in *ClassA* and *ClassB*. In some cases, a pattern needs information that is not concerned by the pattern. For example, *P1* uses the information of the existing system *ClassX* in order to document the modification tasks required for *ClassC*, *ClassD*, and *ClassE*. A similar situation appears when dealing with *P2*; the pattern uses the information of *ClassC* which should already be generated by *P1*.

In this way, we can see how certain patterns may depend on other ones defining a partial order of applying the patterns. This can also be understood by the fact that in a typical maintenance situation, some maintenance tasks may depend on other ones. The result of applying the maintenance patterns on an existing system can take two forms. In some cases, the pattern adds a new element to the system. For example, pattern *P1* adds the new class *ClassC* to the system. In other situations, the pattern may just modify an existing system element. For example, pattern *P2* modifies class *ClassA* by probably adding a new operation to the class or by adding a new code snippet to an existing operation.

Building the system consisting of several maintenance patterns (as in Figure 7) can be regarded as a three phase process. First, modification points need to be identified. Second, related points should be tied up under the same pattern. Third, the relationships among the patterns of the system should be recognized to provide the documentation for the whole maintenance process.

In the building process, we can rely on earlier work about concern architectures [13, 14]. For example, in [13], it is shown how concern architectures might ease documenting framework specialization using patterns. When considering maintainability as a concern [15], we can talk about *maintainability concern architectures*. Typically, each concern tackles a specific area of interest in the maintenance activity. Concerns can, in turn, be composed of smaller units of interest. In this work, we treat each of these small units as a maintenance pattern.

Figure 8 shows the maintainability concern architecture of a specific software system. The architecture consists of three concerns *X*, *Y*, and *Z*. Individual concerns may further be decomposed into several smaller sub-concerns. Concern *X*, for example, is decomposed into two sub-concerns *X1* and *X2*. Concern *Z* consists of two smaller units of interest. These two units are shown in the figure as two patterns. Concerns *Y* and *Z* are overlapping meaning that changes in concern *Y*, for example, might also affect concern *Z*. This means that the maintainer needs to pay more attention when maintaining such overlapping concerns. The architecture may also define inter-concern relationships. This is translated into dependencies when constructing the patterns. For example, in the right part of Figure 8, we see the pattern system that should be constructed; the dependencies between the pattern are shown using dashed arrows.

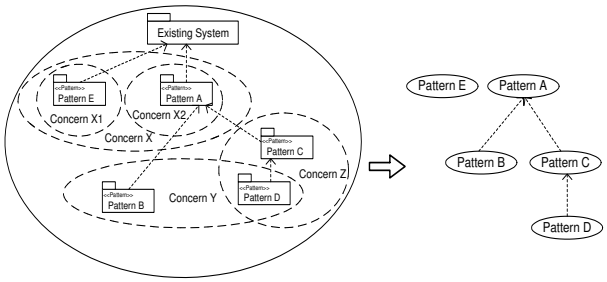

Figure 8. Maintainability concern architectures.

Nevertheless, a typical maintainability concern architecture does not cover all possible maintenance tasks of a system. In fact, maintenance patterns are mostly suitable for adaptive maintenance but are less used for managing other maintenance tasks such as correcting faults or improving system attributes. As a result, a number of code points may remain uncontrolled by maintenance patterns. However, even in this situation, maintenance patterns may prove to be useful. This is obvious when the execution of a maintenance action, that is not controlled by a pattern, breaks other pattern controlled decisions. For example, the maintainer may attempt to improve a certain attribute of a system by splitting an existing class into two new ones. If the original class is referred by a maintenance pattern, the binding between the corresponding pattern role and that class becomes unresolved. As a result, a warning for unresolved element is reported by the pattern.

5 JavaFrames maintainability concern architecture

In this section, we consider applying our approach on JavaFrames system itself. This can be considered as an example of adaptive maintenance.

The pattern engine, an integral part of the JavaFrames system, has been designed to support different domains. In JavaFrames terminology, we refer to these domains as *semantics*. In the same domain, there can be different semantics types. For example, in the Java domain, there are different semantics types for Java classes, Java methods, etc. Currently, the tool supports Java and UML semantics. However, it can be extended to support other semantics such as C++ or XML semantics.

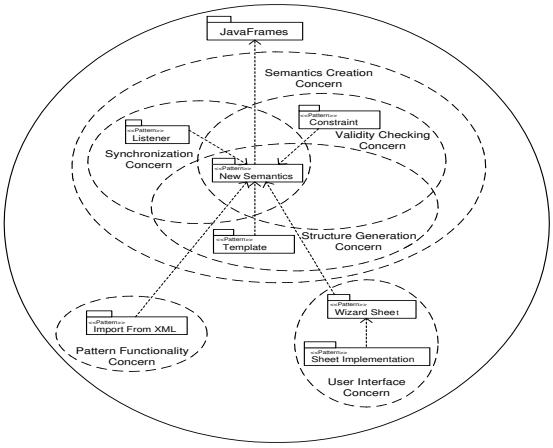

Figure 9. JavaFrames maintainability concern architecture.

Figure 9 shows the maintainability concern architecture of the JavaFrames system. Our interest in the system is from the angle of adapting it to new semantics. The figure defines three concerns:

- Semantics Creation Concern: encapsulates the aspects of creating a new semantics type. It consists of three overlapping sub-concerns that have one pattern in common. The pattern specifies how to create a new semantics type. The three sub-concerns, in turn, are:
  - Synchronization Concern: synchronizes the pattern model with the real world of concrete elements. It is composed of a listener pattern that specifies how to listen to changes in the real world that may affect the current role bindings. Let us suppose, for example, that a Java class role is bound to Java class *A*. If class *A* happens to be renamed or deleted, the associated binding should be broken.
  - Validity Checking Concern: deals with defining possible constraints on the newly created semantics type. It consists of a pattern that generates code for constraint implementation. An example constraint is enforcing the right access permissions of files.
  - Structure Generation Concern: abstracts the properties of the newly created semantics type. It defines a pattern that guides the user how to implement new templates for new type. An example template is the directory information of files.
- Pattern Functionality Concern: defines rules how to extend the existing pattern functionality so that newly generated semantics types are supported. The enclosed pattern, for example, specifies the steps required for creating the right pattern role when reading XML representation of the corresponding type.
- User Interface Concern: encapsulates the steps required for generating wizards for providing concrete elements that play the role of the new type. The 'Wizard Sheet' pattern specifies steps how to define new wizard sheets whereas the 'Sheet Implementation' pattern defines tasks to provide concrete implementations of the wizard sheets.

Figure 10 shows an example maintenance session for adding a new semantics type to the set of existing types. The left view of the figure shows the hierarchical structure of the deployed patterns. The patterns have been identified in the maintainability concern architecture presented in Figure 9. The middle bottom view shows the currently performed bindings. The right view presents the maintenance task to be performed. The upper part of the view indicates the title of the task whereas the bottom part displays a detailed task description. The upper view shows the integrated editor which displays the outcome of performing the maintenance task.

6 Related work

In this section, we investigate related approaches that try to solve maintenance problems at both the methodology and the tool support levels. We start by discussing related methodologies, and then compare our environment with several related maintenance tools. After that, we consider maintainability from the point of view of aspects. Finally, we compare maintenance patterns to related approaches.

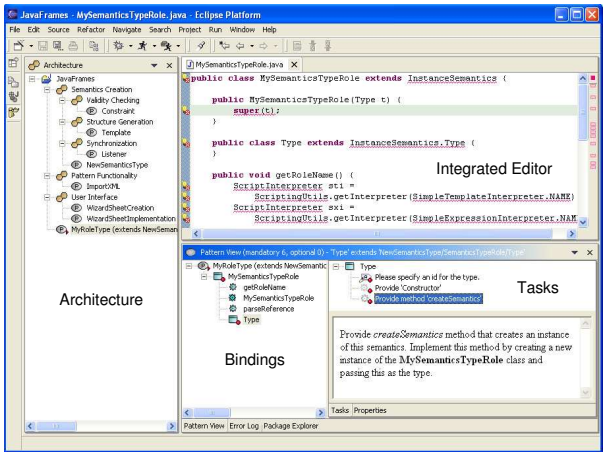

Figure 10. An example maintenance session.

6.1 Work related to maintenance patterns

There is a lot of work done concerning different kinds of patterns. Technically, maintenance patterns resemble design patterns [10], but they are usually system-specific. Although our maintenance patterns may emerge in the design phase, their purpose is to support software maintenance. There are other patterns occurring especially in the maintenance phase. AntiPatterns describe bad programming styles that can be found during maintenance and that should be replaced with better styles [6]. In addition, there is ongoing work concerning identification of design patterns afterwards during software re-engineering or maintenance [7].

Our maintenance patterns are related to re-engineering patterns [8]. Re-engineering patterns provide guidelines to some reverse engineering and re-engineering tasks. They describe common situations that may occur during re-engineering and give advice how to manage these problematic situations. Compared to maintenance patterns, re-engineering patterns are meant to be in post-delivery situations of software, they cannot be applied in the design phase like maintenance patterns. In addition, re-engineering patterns are usually common to all systems whereas maintenance patterns are more system-specific.

Concerning maintenance, there are also lifecycle maintenance patterns or evolution patterns [4]. However, lifecycle maintenance patterns are not system-specific, because they describe typical evolutionary changes that are common to several software systems. Furthermore, lifecycle maintenance patterns try to find common general rules how systems evolve, while re-engineering patterns describe problematic situations that might occur during system evolution and present possible concrete solutions.

6.2 Work related to maintenance tools

There are several maintenance tools for different purposes. Some tools are meant for searching interesting code pieces (for example [19]). There are also tools that enable program representation as hypertext (for example [9, 18]). In these tools, for example, a function name (in function call) provides a link to the corresponding function implementation.

As shown earlier, we have used JavaFrames for documentation purposes needed in software maintenance. However, maintenance tools supporting documentation are typically such that they produce new documentation (e. g. different graphs) according to source code. Their purpose is to clarify the dependencies between different code elements. Together with these graphs, the program becomes easier to understand. The JavaFrames approach is different, because documentation is needed in order to link together code pieces that belong to the same maintenance task. Thus, concerning the provided links, JavaFrames (or actually the maintenance patterns) has similarities to such tools that include the hypertext feature.

There are tools that support change management by locating features in the program code [24]. These features can be domain-specific concepts or functional properties of the system. To make modifications to the features, their location should be identified in the code. Some of these tools, such as presented in [11], apply a static approach. Tracing of related code pieces is based on graphs telling the dependencies between program elements, for example, in procedure calls and data flow.

Searching for feature location can also be implemented dynamically [23, 25, 26]. Searching for the code points where a certain feature is implemented is based on running the program to be changed. The user may run a set of tests that are divided into invoking tests and excluding tests [23]. The execution traces are analyzed to find the code components that were executed in the invoking tests but not in the excluding tests. The set of tests must be selected very carefully, because poorly selected tests will lead to inaccurate selection of the components.

A program feature may correspond to a maintenance task, if a maintenance task is a modification to an existing feature. To complete the task, the involved code pieces should be known. In the aforementioned tools, this identification is performed afterwards and repeated when necessary. In JavaFrames (and especially via maintenance patterns), the documentation, preferably made during software design, include the information that the other tools try to find. Thus, the task of the other tools can be considered as some kind of maintenance pattern mining.

6.3 Work related to aspects and concerns

Aspect-oriented software development (AOSD) supports several concerns that cut across the most evident decomposition of the system [2]. A typical system consists of several kinds of concerns such as business logic, performance, data persistence, logging and debugging [15]. In addition, it is possible to detect development-process concerns, for example, comprehensibility, maintainability, traceability, and evolution ease. In this paper, we are especially interested in maintainability concern.

Maintainability can be considered as a concern. However, at a finer-granular level, we can consider individual maintenance tasks as crosscutting concerns in the system to be maintained. We use maintenance patterns to recognize and manage such concerns. Concern architectures are considered in [13, 14]. In the terminology of [13], maintainability can be managed by identifying maintenance patterns and their relationships. The pattern relationships in [13] are used to define a partial order for applying the patterns. In addition, the relationships identify the parts where modifications to program elements referred by a pattern can affect other elements bound by the overlapping patterns.

Maintenance patterns are closely related to concern graphs [20]. In concern graphs, related program features (or code pieces) form a concern that is represented as a graph. Code pieces forming a concern belong to the same maintenance task. In the same conceptual way, maintenance patterns collect together the related pieces of source code. However, concern graphs differ from maintenance patterns in that concern graphs are extracted from the existing source code repeatedly, and they are not permanent like the information in maintenance patterns. Moreover, concern graphs do not support maintenance documentation.

6.4 Comparison of maintenance patterns to related approaches

The tools presented in Figure 11 are related to each other, because they all have the common purpose, i.e. change management. Our approach is referred in the figure as maintenance patterns (MP). Most of the methods support identification of feature locations. Maintenance patterns are mostly related to concern graphs, because both methods support concerns. We can consider concerns as a mechanism that brings and links together associated pieces such as patterns in our approach. In this standpoint, hypertext links can serve the same purpose, and thus, the HyperSoft tool [18] has also been checked off in that row.

The other considerations discussed in previous subsections are also collected in Figure 11. Maintenance patterns can be characterized such that although their purpose (change management) is similar to other tools, they support the purpose in a different way. For example, they provide permanent information about related code points, whereas other tools extract information repeatedly. Moreover, maintenance patterns support maintenance consideration already in the design phase.

Feature	MP	[20]	[11]	[18]	[25]	[26]
static (S) / dynamic (D) information	S	S	S	S	D	D
extracted (E) / permanent (P) information	P	E	E	E	E	E
only maintenance (M) / also design (D) phase consideration	D	M	M	M	M	M
concern support (link information)	x	x		x		
feature location support	x	x	x		x	x

Figure 11. Comparison of maintenance patterns (MP) to other existing tools.

7 Conclusions

In this paper, we have introduced maintenance patterns to support software documentation especially for maintenance purposes. Maintenance patterns aid in linking together those program parts that need modification due to the same larger maintenance task. A program piece may belong to several maintenance tasks creating a link between the corresponding maintenance patterns. Documenting these links support change impact analysis.

We show how maintenance patterns can be applied using JavaFrames even though JavaFrames is originally not built as a maintenance tool, but is meant for application development purposes. For example, the task list property supports maintenance tasks directly, because maintenance tasks can typically be divided into smaller tasks that can be performed in a certain order.

So far, we have only limited experience in applying the approach presented in this paper to real maintenance situations. A real case study is the next action of future work, and for this purpose we have tool support available. In this way, we can find out how appropriate and practical the discussed approach is and how it scales up to real world examples.

As another future work, maintenance patterns can be used to provide information about maintenance history. We can collect useful information about those code points that need recurring modifications and/or those code points that belong to several maintenance patterns. It can be assumed that typically there are code points that meet frequent change actions and others that are more stable. Via maintenance patterns, we could find these different points. In the future, there could even be a property which can be used to visualize maintenance patterns and the evolution of the system. Another future research direction is to investigate how

the use of maintenance patterns could ease system testing. We can assume that using maintenance patterns it is possible to select the test cases more precisely. Particularly in regression testing, we need not test such parts of the system that are not involved in the implemented change.

Acknowledgments

This work is funded by the National Technology Agency of Finland (Tekes), Nokia Mobile Phones (NPM), Nokia Networks (NET), Plustech, Almare, Profit, and the Academy of Finland. The authors would also like to thank Kai Koskimies, Tommi Mikkonen, and Mika Katara for their comments.

References

[1] IEEE standard for software maintenance. IEEE Std 1219, 1992.

[2] Communications of the ACM. Special issue on Aspect-Oriented Programming, 44:10, 2001.

[3] Eclipse WWW site. Available at http://www.eclipse.org, 2003.

[4] E. J. Barry, C. F. Kemerer, and S. A. Slaughter. On the uniformity of software evolution patterns. In *Proceedings of ICSE'03*, pages 106–113, Portland, Oregon, May 2003.

[5] K. H. Bennett. Software Evolution: past, present and future. *Information and Software Technology*, 38(11):673–680, January 1996.

[6] W. J. Brown, R. C. Malveau, H. W. McCormick III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, Erewhon, NC, 1998.

[7] W. C. Chu, C. W. Lu, C. Chang, and Y. C. Chung. Pattern-based software re-engineering: a case study. *Journal of Software Maintenance: Research and Practice*, 12(2):121–141, 2000.

[8] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[9] B. Freitag. A hypertext-based tool for large scale software reuse. In *Proceedings of CAiSE'94*, pages 283–296, Utrecht, The Netherlands, June 1994.

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

[11] S. Gwizdala, Y. Jiang, and V. Rajlich. JTracker — a tool for change propagation in Java. In *Proceedings of CSMR'03*, pages 223–229, Benevento, Italy, March 2003.

[12] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Generating application development environments for Java frameworks. In *Proceedings of GCSE'01*, pages 163–176, Erfurt, Germany, September 2001.

[13] I. Hammouda, M. Pussinen, M. Katara, and T. Mikkonen. UML-based approach for documenting and specializing frameworks using patterns and concern architectures. In the 4th workshop of AOSD Modeling with UML, October 2003. San Francisco, California.

[14] M. Katara and S. Katz. Architectural views of aspects. In *Proceedings of AOSD'03*, pages 1–10, Boston, Massachusetts, March 2003.

[15] R. Laddad. I want my AOP!, part 1: separate software concerns with aspect-oriented programming. Available at: http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html, 2002.

[16] T. Matsumura, A. Monden, and K. Matsumoto. A method for detecting faulty code violating implicit coding rules. In *Proceedings of IWPSE'02*, pages 15–21, Orlando, Florida, May 2002.

[17] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proceedings of ECOOP'98*, pages 355–382, Brussels, Belgium, July 1998.

[18] J. Paakki, J. Koskinen, and A. Salminen. From relational program dependencies to hypertextual access structures. *Nordic Journal of Computing*, 4(1):3–36, 1997.

[19] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[20] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of ICSE'02*, pages 406–416, Orlando, Florida, May 2002.

[21] N. W. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):303–310, 1987.

[22] E. B. Swanson and E. Dans. System life expectancy and the maintenance effort: Exploring their equilibration. *MIS Quarterly*, 24(2):277–297, 2000.

[23] T. Technologies. Telecordia software visualization and analysis toolsuite (xSuds). User's manual, Chapter 12, 1998. Available at: http://xsuds.argreenhouse.com/html-man/xvue.html.

[24] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 65(2):105–114, 2003.

[25] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique program comprehension. In *Proceedings of ICSM'96*, pages 312–318, Monterey, California, November 1996.

[26] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Locating program features by using execution slices. In *Proceedings of ASSET'99*, Richardson, Texas, March 1999.