A Meta-model for Language-Independent Refactoring[1]

Sander Tichelaar[*], Stéphane Ducasse[*], Serge Demeyer[+] and Oscar Nierstrasz[*]

[(*)]*Software Composition Group, IAM, Universität Bern*
Neubrückstrasse 10, CH-3012 Berne, Switzerland
{tichel,ducasse,oscar}@iam.unibe.ch www.iam.unibe.ch/~scg

[(+)]*Lab on Reengineering, University of Antwerp*
Universiteitsplein 1, B-2610 Wilrijk, Belgium
Serge.Demeyer@uia.ua.ac.be. win-www.uia.ac.be/u/sdemey

1. Proceedings ISPSE 2000, IEEE, 2000, pp. 157-167.

Abstract

Refactoring —transforming code while preserving behaviour— is currently considered a key approach for improving object-oriented software systems. Unfortunately, all of the current refactoring tools depend on language-dependent refactoring engines, which prevents a smooth integration with mainstream development environments. In this paper we investigate the similarities between refactorings for Smalltalk and Java, derive a language-independent meta-model and show that it is feasible to build a language-independent refactoring engine on top of this meta-model. Our feasibility study is validated by means of a tool prototype which uses the same engine to refactor both Smalltalk and Java code. Using our approach we minimize the language-dependent part of refactoring tools, providing a standard way for programmers and tools to perform refactorings no matter what language they work in.

1. Introduction

Refactoring is defined as "changing a system to improve its internal structure without altering its external behaviour" [5]. Especially when automated by a tool, refactoring is an easy, quick and safe way to improve software systems at the code level. Recently, refactoring has been gaining widespread acceptance, as for example illustrated by the increasing popularity of the Refactoring Browser [7] (an almost ubiquitous Smalltalk tool), the growing success of Fowler's refactoring catalogue [5] and the explicit integration of refactoring into a software development method (namely Extreme Programming [9]). Due to this growing demand, one may safely assume that refactoring tools will soon become part of mainstream software development environments.

Surveying existing software development environments, we perceive alongside the standard code editors a whole range of program auditing facilities. These facilities support developers in understanding existing programs (e.g., extracting UML diagrams, generating documentation) and in detecting potential problems (e.g., gathering performance profiles, measuring code attributes). Of course, a tight integration between these auditing facilities and refactoring tools is highly desirable as the former provides the necessary input for the latter. Most commonly, integration in software development environments is achieved by means of a repository: a shared database accumulating all knowledge about the software system under development. Thus, for refactoring tools to become part of mainstream software development environments, the most natural way would be to adhere to a repository architecture.

Such a repository architecture requires a central data model (the so-called *meta-model*) which —to deal with the multiple languages these environment typically support — should be highly language independent and contain sufficient information to represent refactorings. While there is sufficient proof that a refactoring tool can be built for almost any object-oriented language (Smalltalk [10], Eiffel [1], Java [11][12] and C++ [6][13]) it is yet unknown whether it is feasible to build a language-independent refactoring engine. Support for multiple languages in a refactoring tool is mentioned by Ó Cinnéide [12]. He presents a layered architecture which shields language specifics as much as possible, but so far his tool prototype only supports one language, namely Java.

This paper presents the results of a feasibility study for a language-independent refactoring engine. Based on the list of primitive refactorings, we derive a common meta-model that is sufficient to perform the necessary analysis. We validate our sufficiency claim by means of a tool prototype which uses a single engine based on a common meta-model to refactor both Smalltalk and Java. We have chosen Java and Smalltalk, because these two languages are both mainstream object-oriented languages and they differ sufficiently to make the step to language independence non-trivial. The most significant differences are that Smalltalk has explicit metaclasses, that Java has a special interface concept, and that Java is statically typed while Smalltalk is dynamically typed.

After presenting the context of our work in section 2, we present a meta-model for describing object-oriented systems that enables us to describe refactorings at a language-independent level (section 3). We categorize the refactorings from the perspective of this model (section 4) and discuss two refactorings, Add Class and Rename Method, in detail as an illustration of what issues typically arise (section 5). The research has been verified by building a prototype that applies a non-trivial sequence of refactorings on a similar software system in both Java and Smalltalk (section 6). In this case-study all described refactorings are used. We finish with a discussion, related work and conclusion.

2. Experimental Set-up

The work presented in this paper has been done in the context of the FAMOOS project [14], a European ESPRIT project on reengineering object-oriented software. In this project we have developed a tool environment called Moose [4] which has the goal to support reengineering tasks such as metrics, visualisation and reorganisation, and which is based on a repository architecture. One of the goals — which is the main topic of this paper — was to integrate refactorings. We introduce here the design goals of the model, which defines the framework in which the rest of the paper needs to be viewed.

The repository and its underlying model have been developed with the following goals in mind:

- *Support for multiple languages*. We had to deal with different object-oriented languages (C++, Java, Smalltalk, Ada) and did not want to rewrite our tools for each of those languages.
- *Minimalisation of information.* Our aim was not to cover all aspects of all languages, but rather to capture the common features that we needed for reengineering activities.

To fulfil these goals the model describes software systems at the so-called *program entity level* as opposed to the abstract syntax tree level. The most detailed information the model represents concerns local variables and which method calls which method and accesses which attribute, but we do not analyse the exact control flow within methods. This allows us to reason at a level which is sufficiently abstracted from language-specific details, but which is sufficiently detailed to support the analysis we need to perform.

We claim that this level of information is sufficient to support most of the primitive refactorings on a language-independent level. The refactorings we support are shown in table 1 together with their pre- and postconditions. They are what Opdyke [6] calls *low-level* refactorings, i.e. primitive program transformations for adding, removing and renaming entities and moving entities within their inheritance hierarchies. These low-level transformations can be combined to perform more complex transformations, called *high-level* refactorings, for instance to introduce design patterns [12][5]. The high-level refactorings are outside the scope of this paper. They are typically a combination of low-level refactorings and therefore have not much to do with language issues that are handled on the lower level.

Consequent to the choice not to model information that requires a thorough analysis of method bodies, we do not support refactorings that need this kind of information. Examples include Extract Method and Move Method [5]. The last kind of refactorings that are not covered in this paper are those that are relevant for only a single language. These include Change Type and Move Class (between packages).

Note that changing code can be done in a safe way only when the information about a software system is 100% complete. Anything less might result in a non-working or wrongly behaving system. Although extracting all information is not always a trivial task we assume in this paper we have all required information readily available.

3. A Language meta-model for Refactoring

In order to be able to check the preconditions of table 1 and to analyse which code changes need to be undertaken for every supported refactoring at a language-independent level, we have developed the FAMIX model. FAMIX provides for a language-independent representation of object-oriented source code. It is an entity-relationship model that models object-oriented source code at the *program entity level*. figure 1 shows the core entities and relations. The core model

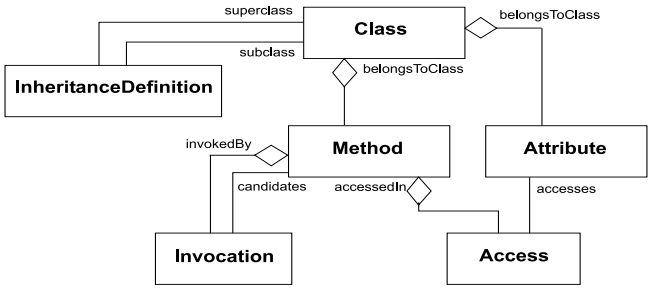

Figure 1 The core of the FAMIX model

specifies the entities and relations that are extracted immediately from source code. It consists of the main object-oriented entities, namely Class, Method and Attribute. In addition there are the associations InheritanceDefinition, Access and Invocation. An Access represents a Method accessing an Attribute and an Invocation represents a Method calling another Method. These abstractions are needed for dependency analysis. Note that we model statically determinable invocations. The actual invocations at runtime can be to any method that is polymorphically equivalent to the statically

determined method. The complete model, which is set up as an object-oriented hierarchy, consists of more entities, such as functions and formal parameters. Additionally to the en-

Table 1: Overview of the refactorings supported by our approach

Refactoring	pre-condition	post-condition
Add Class (*classname*, *package*, *superclasses*, *subclasses*)	• no class exists with *classname* in the same scope • no global variable exists with *classname* in the same scope • subclasses are all subclasses of all superclasses • [Smalltalk] superclasses must contain one class • [Smalltalk] superclasses and subclasses cannot be metaclasses	• new class is added into the hierarchy with *superclasses* as superclasses and *subclasses* as subclasses. • new class has name *classname* • *subclasses* inherit from new class and not any more from *superclasses*
Remove Class (*class*)	• *class* has no attributes or its attributes are not referenced • idem for methods • *class* does not implement abstract methods from its superclass hierarchy • [Smalltalk] *class* cannot be a metaclass	• *class* is removed (including non-referenced attributes and methods) • superclasses of *class* are now superclasses of its subclasses • [Smalltalk] corresponding meta-class is deleted as well
Rename Class (*class, new name*)	• no class exists with *new name* in the same scope • no global variable exists with *new name* in the same scope • classes that refer to *class* do not have any (inherited) variable with *new name* • [Smalltalk] a metaclass cannot be renamed independently of the class it represents	• *class* has *new name* • all references (types, casts, class method calls, superclass references) are updated with the new name • [Java] constructors are updated with the new name • [Java] casts to *class* have been updated • [Smalltalk] the corresponding metaclass of *class* has been renamed as well
Add Method (*name, class*)	• no (inherited) method with signature derived from *name* exists in *class*	• *class* has a method called *name* with an empty body or is abstract if *class* represents a Java interface
Remove Method (*method*)	• *method* has no static candidate invocations	• *method* is removed from its containing class
Rename Method (*method, new name*)	• no method exists with the signature implied by *new name* in the inheritance hierarchy that contains *method* • [Smalltalk] no methods with same signature as *method* outside the inheritance hierarchy of *method* • [Java] *method* is not a constructor	• *method* has *new name* • relevant methods in the inheritance hierarchy have *new name* • invocations of changed method are updated to *new name*

Table 1: Overview of the refactorings supported by our approach

Refactoring	pre-condition	post-condition
Pull Up Method (*method, superclass*)	• *method* should not access attributes or methods from its containing class • *superclass* does not contain a method with the same signature as *method* • *method* cannot have super references to *superclass* • pulled up method should not hide implementations higher up in the hierarchy from referring subclasses of *superclass* • [Java] *method* is not a constructor • [Java] non-empty method cannot be pulled to an interface	• *method* defined in *superclass* • *method* not defined in original containing class
Push Down Method (*method*)	• *method* is not invoked in or through its containing class • direct subclasses of the containing class of *method* do not contain a method with the same signature already. • [Java] *method* is not a constructor	• *method* not defined in original containing class • *method* defined in subclasses of the containing class
Add Parameter (*name*, *method*)	• *method* does not have a parameter with *name* already • *method* does not have a local variable with *name* already • no method exists with the signature implied by *name* in the inheritance hierarchy that contains *method* • [Smalltalk] containing class does not have an attribute with *name* • [Smalltalk] no methods with same signature as *method* outside the inheritance hierarchy of *method*	• *method* and all relevant methods in the inheritance hierarchy have an extra parameter with *name* • invocations of *method* are updated to invoke it with an extra parameter with a default value
Remove Parameter (*parameter*)	• parameter is not referenced in the containing, any overriding or overridden method • no method exists with the signature implied by removing *parameter* in the inheritance hierarchy of the containing method • [Smalltalk] no methods with same signature as *method* outside the inheritance hierarchy of *method*	• *method* and all relevant methods in the inheritance hierarchy have *parameter* removed • invocations of *method* are updated to invoke it without *parameter*
Add Attribute (*name, class*)	• no (inherited) attribute with *name* exists in *class* • subclasses do not contain attribute with *name*	• *class* has attribute named *name*
Remove Attribute (*attribute)*	• *attribute* is not accessed	• *attribute* is removed from its containing class

Table 1: Overview of the refactorings supported by our approach

Refactoring	pre-condition	post-condition
Pull Up Attribute (*attribute, superclass)*	• any attribute in the superclass and its subclasses with the same name as *attribute* has the same type as *attribute* • *attribute* will not hide another attribute with the same name	• *superclass* contains *attribute* • all attributes in the subclasses of superclass with the same name and type as *attribute* have been removed
Push Down Attribute (*attribute)*	• *attribute* is not referenced or accessed through its containing class • the direct subclasses of the containing class do not contain an attribute with the same name as attribute	• *attribute* is removed from its containing class • all subclasses that need it (i.e. that have a reference to the attribute somewhere in its hierarchy) define an attribute with the same name and type as *attribute*

tities themselves, FAMIX defines for every entity a set of attributes. A Method, for instance, has attributes such as `signature` and `isAbstract`. The complete specification of the model can be found in [3].

The different supported languages need to be mapped to FAMIX. For Java [8], C++ [16] and Ada [15] we have described those mappings. The Smalltalk mapping is work in progress. The goal of the mappings is to be able to treat all languages similarly. However, as shown in table 1, in some cases the difference in semantics of a concept in two languages cannot be ignored. An example is the mapping from Java interfaces to FAMIX classes. Additionally, we need to store language-specific information to be able to do the necessary language-specific analysis.

Certain design choices for the meta-model have special impact on refactorings and refactoring analysis for Smalltalk and Java, and therefore require special attention. Note that they are discussed in more detail in section 7.

FAMIX supports types

Static type information is important information to store for languages that support it such as Java and C++. Dynamically typed languages such as Smalltalk are covered by storing the most general type (Object) wherever needed.

FAMIX supports multiple inheritance.

Naturally this covers languages with multiple inheritance such as C++ and single inheritance in, for instance, Smalltalk. Java is covered by interpreting Java interfaces as abstract classes.

One might wonder why we came up with our own model in the first place. One reason is that when we started we did not find any model that adequately modelled source code in the way we needed it to. Different source code models exist [17], but typically they do not have clearly defined mappings to different languages. Another reason is that models such as UML [18] are directed towards object-oriented analysis and design rather than source code representation. Especially concepts such as invocations and accesses are hard to model using UML. This issue is extensively discussed in [2].

4. Language-independent refactoring

The meta-model described in section 2 allows us to reason about the refactorings introduced in table 1 at a language-independent level. To see if a refactoring is language-independent, or rather *how language independent a refactoring is*, we take FAMIX as a reference. We categorize the analysis, i.e. checking the preconditions and determining what needs to be changed, in the following way:

- *complete reuse.* The analysis is completely language-independent, meaning that the model can give a conclusive answer to the question whether a certain precondition holds or what exactly needs to be changed, for all supported languages. An example of this case is that if a class is to be renamed, the new name may not already exist in the context of that class. This rule applies for both Smalltalk and Java.
- *interpretation issues.* The analysis can be performed based on the model, but needs to be interpreted differently for different languages. For instance, types are checked in the model, but are of no relevance for Smalltalk which is dynamically typed.
- *language-specific analysis.* Parts of the analysis need language-specific information (for instance, an extra check needs to be made in case a class represents a Java interface). Typically this information is available in the model through language extensions [8][15][16].
- *front-end issues.* The front-ends, i.e. the actual code changers, are naturally specific for the language they change the code of. Apart from the trivial issues of the syntax of the language, sometimes analysis must be performed to apply changes according to the specific language rules. For instance, although both Java classes and interfaces are represented as classes in the model, in some cases they need to be treated differently at the code level.

Typically a refactoring does not fall into only one category as its preconditions and latent code changes often are partly language independent, partly they need a language-specific interpretation, and yet other parts are language specific. In ta-

Table 2: Language dependency issues for refactorings in Java and Smalltalk

Refactoring	interpretation issues	language-specific analysis	front-end issues
Add Class		Java interfaces, Smalltalk metaclasses	Java interfaces
Remove Class	type related analysis	Smalltalk metaclasses	Java interfaces
Rename Class	type related analysis	Smalltalk metaclasses, class methods, Java constructors, Java casts	Java interfaces
Add Method			Java interfaces, Java constructors, default types
Remove Method			Java abstract methods
Rename Method		Java constructors, lack of static type information	
Pull Up Method		Java constructors, lack of static type information, Java interfaces	
Push Down Method		Java constructors, lack of static type information	
Add Parameter		lack of static type information	default types
Remove Parameter		lack of static type information	
Add Attribute		global variables	default types
Remove Attribute			
Rename Attribute		global variables	
Pull Up Attribute	Java hiding		
Push Down Attribute			

ble 2 we present the categorization of our refactorings using short descriptions of how each refactoring depends on language specific issues.

Java interfaces

Interfaces in Java require special rules to be observed. One cannot, for instance, pull up a non-abstract method to an interface. For the class refactorings and the Add Method refactoring the Java code changing front-end needs to know if it is dealing with Java classes or Java interfaces.

Smalltalk metaclasses

FAMIX interprets Smalltalk classes and Smalltalk metaclasses as classes. However, the semantics of Smalltalk imposes certain rules. Every class in Smalltalk has a metaclass associated with it. Metaclasses do not have an explicit name and cannot be added or removed independently of the normal classes they represent.

Static vs dynamic typing

Due to the type-awareness of FAMIX, and the mapping to FAMIX of statically typed Java and dynamically typed Smalltalk, three phenomena can be observed:

- *Type related analysis.* In several refactorings there exists analysis for dealing with typed information (for instance, in the case of a Rename Class refactoring, the types of the attributes that have this class as a type need to be changed). For Smalltalk much of that analysis is unnecessary. The query for all attributes with a certain type will return the empty set and this is known beforehand. Note that this does not make the refactoring language-independent. It just means that analysis is done

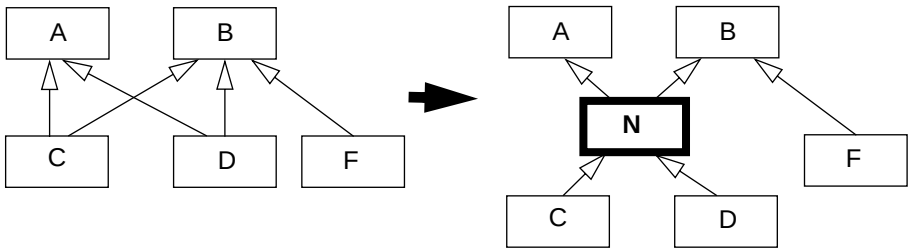

Figure 2 Add Class refactoring with *classname* N, *superclasses* A and B and *subclasses* C and D

that is unnecessary for Smalltalk: preconditions will not be violated and it will not result in any changes in the Smalltalk sources.

- Due to *the lack of static type information in Smalltalk*, invocations to a certain method name cannot be tracked to one implementation (or at least one hierarchy with implementations). To apply the Rename Method refactoring safely, the model needs to be checked for implementations outside of the inheritance hierarchy of the target implementation. The Add Parameter and Remove Parameter refactorings in Smalltalk suffer from the same problem, because in Smalltalk adding and removing a parameter require a method name change.
- *Default types.* Several creational refactorings (Add Method, Add Attribute and Add Parameter) need to provide type information for Java. The solution we have chosen is to assign default types (Object for new attributes and parameters, void for method return types). Another solution would be to ask the user for a type and ignore this information in the Smalltalk case.

Class methods

Due to the different way of representing class methods — instance methods of metaclasses in Smalltalk and static methods in Java — class method calls need to be gathered from the model in a language-specific way.

Java constructors

In Java constructors are a special kind of method. Special rules apply, for example, that a constructor has to have the same name as its class, it does not have a return type, and the syntax to invoke it is different from a normal method invocation. In FAMIX Java constructors are represented as normal methods. Therefore, to cover constructors extra analysis needs to be performed to ensure the naming conventions are adhered to, and the code changer needs to interpret invocation information differently. Also some refactorings cannot be applied to constructors such as Pull Up Method and Push Down Method. And renaming a Java constructor can only be done in the context of a Rename Class refactoring.

Global variables

New attributes in Smalltalk cannot have the same name as a ‘global’ (i.e. global classes and global variables), because this might hide these globals in the scope of the new attribute. In Java types and attribute names do not interfere.

5. Two refactorings in detail

In this section we discuss two refactorings in more detail. We present the Add Class refactoring and the Rename Method refactoring with their definitions, their preconditions and a discussion of issues regarding language-independence and mapping of information to FAMIX. We have chosen these two refactorings, because they cover some typical problems and illustrate the complexity of the language independent analysis. Trivial preconditions like “a class should really be a class” are not mentioned. For both refactorings we compare our approach with language-specific approaches by Werner [11], Roberts[10] and Opdyke [6]. These PhD theses describe refactorings including their pre- and postconditions for Smalltalk, Java and C++ respectively.

Add Class (*classname, package, superclasses, subclasses*)

> Inserts a new class with name *classname* in package *package* where *superclasses* are the superclasses of the new class and *subclasses* are subclasses of all *superclasses* that have to become subclasses of the new class (see figure 2).

Typically this is a simple refactoring, because the new class is not referenced yet, so no relationships need to be updated and only name clashes need to be checked. However, abstractness of classes and multiple superclasses need to be taken into account.

Dealing with abstract classes. When the new class inherits abstract methods without implementing them, it must be declared abstract. The model contains the information to determine this. However, for Smalltalk the analysis is unnecessary, because in Smalltalk abstractness of classes is implicit.

Single versus multiple superclasses. Multiple inheritance can be easily supported if the precondition that all *subclasses* inherit from all *superclasses* is fulfilled. Inserting a class in the middle will have no impact on the outside behaviour, because the new class does not add, overwrite or hide any behaviour and the existing classes will still inherit from the

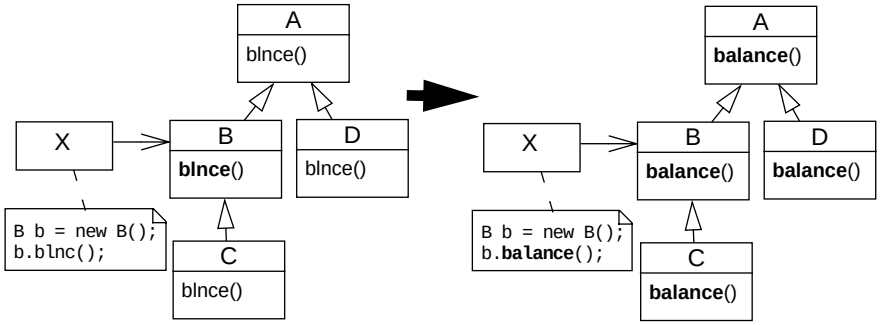

Figure 3 Rename Method refactoring renaming `blnc` in class B to `balance`

same set of classes. Smalltalk's single inheritance is supported by this scheme and also the Java interface concept. However, for Java additional analysis is needed to determine if the new class needs to be an interface or a class.

Classes and files in Java. A new class in Java typically needs a new file to be created as well. This is transparently taken care of by the Java front-end.

Note that we do not deal with inner classes and that we currently do not cope with constructor chaining and array instantiations in Java.

Preconditions

Language-independent preconditions

1. No class with the same unique name already exists
2. No global variable with the same unique name already exists
3. All *subclasses* are subclasses of all *superclasses* or no subclasses are specified

Language-dependent preconditions

4. *Classname* is a valid name.

Smalltalk-specific preconditions

5. *Superclasses* (and therefore *subclasses*) cannot be metaclasses.

Precondition discussion

The preconditions are mainly language-independent. The language-dependent preconditions are about naming rules and checks to ensure that the refactoring is not applied to 'special' classes that cannot be used in standard ways such as metaclasses in Smalltalk and interfaces in Java.

Some comments concerning the preconditions:

- ad 1. Covers classes in Smalltalk and classes and interfaces in Java. Classes with the same name in different packages are allowed by this rule, because the unique name in FAMIX includes scoping (for Java the containing packages are part of the name).
- ad 2. This precondition is a typical example of a language-independent precondition that does not fit all supported languages, but will always return true for the languages it does not fit. Smalltalk supports global variables but Java does not. Therefore, there will never be global variables in Java and consequently no global variables with the same unique name as the new class.
- ad 3. Necessary condition to handle multiple inheritance in a behaviour preserving way. See comments before.
- ad 5. Smalltalk has explicit metaclasses, which map to classes in FAMIX. Every class has an accompanying metaclass. However, it is not possible to create a metaclass independent of a class and thus to add a class in a metaclass hierarchy.

Related work

For this refactoring the main difference with the language specific approaches by Werner [11], Roberts [10] and Opdyke [6] is that they only support single inheritance. For Smalltalk this just follows the language, for Java and C++ this is done for reasons of simplicity. However, we feel it important to support Java interfaces because of their widespread use, which implies multiple inheritance with the mapping we have chosen.

Rename Method(*method, new name)*

> Renames *method* and all method definitions with the same signature in the same hierarchy. All invocations to all changed methods are changed to refer to the new name (see figure 3).

A method can only be renamed in a behaviour-preserving way if all overriding methods and overridden methods (and all their overriding and overridden methods) are renamed as well. Furthermore, all invocations to *all* changed methods need to be renamed accordingly. In the context of language independence the issues of Java constructors and the lack of dynamic type information discussed in section 4, need to be dealt with.

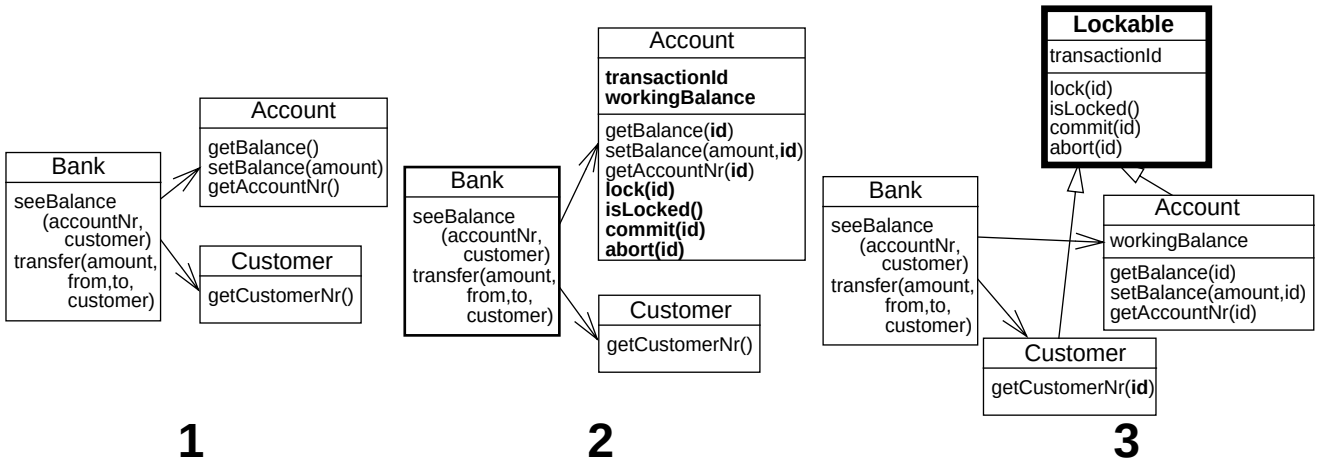

Figure 4 Refactoring scenario which introduces transactional support to a toy banking system

Preconditions

Language-independent preconditions

1. The subclass hierarchies of the classes highest up in the superclass hierarchies of the class containing *method* do not already contain a method with a signature implied by the *new name* and the parameters of *method*.

Language-dependent preconditions

2. *New name* is a valid method name.

Smalltalk-specific preconditions

3. There exists no method with the same signature as *method* outside of the inheritance hierarchy of the class that contains *method*.

Java-specific preconditions

4. When *method* is a constructor the refactoring can not be applied unless in the context of a rename class refactoring.

Precondition discussion

Precondition 1. is language independent, because FAMIX maps language-dependent signatures to a language-independent naming scheme. Also the multiple inheritance scheme covers all supported languages. However, it adds complexity also for languages that only have single inheritance.

Related work

Opdyke [6] and Werner [11] allow for names to be renamed to an already existing name when either the other method is not referenced, or if the methods are *semantically equivalent*. We have chosen a stricter approach, because the first solution, although it works, produces unclear (a same name conveys similar behaviour and a relation between the methods, which in this case does not need to be true at all) and therefore low-quality code. The second option is very hard to check in practice. The Refactoring Browser [7] detects a few cases by checking if two syntax trees are equal with possibly different parameter and local variable names. Additionally to the above arguments the stricter approach is easier to check and to abstract from the specific languages.

Roberts [10] includes an extensive discussion about how dynamic analysis and dynamic refactoring could solve the lack of static type information in dynamically typed languages. In this paper we have limited ourselves to statically available information.

As mentioned before all three approaches only cover single inheritance.

6. Tool Support

We have built a prototype, the Moose Refactoring Engine, that supports the language-independent refactorings described in this paper. It is part of the Moose Reengineering Environment [4], a tool environment for reengineering object-oriented systems. Moose contains a repository, which is based on the FAMIX model. The Refactoring Engine uses the repository to retrieve the required information, perform the needed analysis and calls its so-called language front-ends that act directly on the source code to apply the changes. The Smalltalk front-end uses parts of the Refactoring Browser [7] to change Smalltalk code, the Java front-end currently uses a text-based approach based on regular expressions. Although the text-based approach is more powerful than we initially expected, we plan to move to an abstract syntax tree based approach in the future, because it better abstracts from layout details and is better fit to the more complex code changes such as replacing the third parameter of a method name.

A scenario

The different refactorings implemented in the Moose Refactoring Engine have been tested on two pieces of similar Smalltalk and Java code. A sequence of refactorings has been applied and after every refactoring the adapted software has been tested if it still functions as expected. figure 4 shows the used scenario in a nutshell. A toy banking system (**1**) with just accounts and customers is transformed into a system with transactional support. First the Account class gets transactional support (**2**). New attributes are added (e.g. transactionId), new methods are added (e.g. lock(id)), new parameters are added to existing methods (e.g. id to getBalance()). Method bodies need to be added and adapted as well, which is not covered by the refactorings and therefore done by hand. From **2** to **3** the generic part of the transactional support is lifted into an common superclass for Account and Customer, so that a customer can take part in transactions as well. This step includes the Add Class refactoring (Lockable) and pulling up of attributes and methods. Again some actions need to be taken at the method body level: account specific functionality needs to be separated from transactional functionality before the transactional functionality can be pulled up. This separation could be done using an Extract Method refactoring, but this refactoring is not covered by our model and engine.

This scenario including reversing it from **3** to **1** covers all refactorings of table 1. Although the case study only contains toy code and experiments with real world systems still need to be undertaken, the scenario shows that the approach is applicable to non-trivial, in our view realistic, sequences of refactorings. We are therefore confident that the approach will work for real systems as well.

7. Discussion

We have described refactorings in the light of a language-independent meta-model. Looking at table 2 and at the two refactorings that are discussed in detail, we make the following observations:

Language independence brings useful reusability. Major parts of the refactorings are described and analysed on a language-independent level. Similar concepts in the different languages are treated in a uniform way, resulting in reuse of analysis and reducing the language specifics to only the changes in the source code. However, in some cases the advantages of reuse come at a cost:

- *Increased complexity of algorithms.* To deal with multiple languages the underlying model needs to be general enough to cover the supported languages. For instance, the model supports multiple inheritance, which involves more complexity than would be needed, for instance, for single inheritance in Smalltalk alone.
- *Mapping back to the actual code.* The actual code changes are, naturally, language specific. However, in some cases the concepts that are generalized at the language-independent level (e.g. Java constructors are methods, Java interfaces are classes) need to be mapped back to their language-specific kind, because at the code level they need to be dealt with differently than their 'normal' counterparts. For example, on the code level invocations of Java constructors are different from invocations to 'normal' methods. This implies that the language-specific information about how an entity has been mapped needs to be stored, because it is necessary information when mapping back.
- *Language-independent defaults.* To keep some refactorings as language independent as possible, some defaults are used. Typical examples are types: some refactorings use the most general type, i.e. Object for both Smalltalk and Java. This works well for both languages, although it is clear that support for defining or changing types would be desirable for statically typed languages such as Java.

Not all language differences can be abstracted from.

i.e. most refactorings cannot be completely described at a language independent level. We see the following kinds of issues:

- Standard issues that are apparent in all languages, but need a language-specific interpretation, like if a name of a class is a valid class name *for that language*.
- Issues that are caused by the *mapping* from the language to FAMIX. For example, the meta-model does not know the concept of metaclasses or interfaces. Rules that apply to these specific concepts need to be checked nonetheless and are inherently language specific.
- The most problematic issues are in the *core differences* between the languages. The fact that Smalltalk is dynamically and Java statically typed, means that there is less information available at compile-time. Especially for dependency analysis through invocations and accesses, the type information tells much more precisely which method is invoked or which attribute is accessed. In dynamically typed languages a certain method invocation can be any method with that signature, no matter what class it is defined in. Therefore, some refactorings can only be applied for dynamically typed languages when more severe restrictions are taken into account. An example is the Rename Method refactoring which can only be applied when there is no method with the same signature as the method to be renamed outside of the targeted inheritance hierarchy. Note that the type information for dynamically typed languages can be refined through additional analysis (for instance, using

type inference techniques) [10], but this is outside the scope of this paper.

All in all we can say that the presented model is adequate to represent refactorings for multiple object-oriented languages. The program entity level of information is sufficient for refactorings that do not need detailed information about method bodies. Some language-dependent details, however, must be coped with.

Many design decisions for the model —to apply a language-independent naming scheme including scoping and the different mappings to allow to treat similar constructs in different languages in a similar way— result in language independence and reuse of analysis code. However, especially with the mappings, it is always a trade-off between reuse and complexity. Instead of mapping similar constructs to one representation, the two constructs can be both modelled explicitly. Naturally this decreases problems with differences between the constructs, but it also maked the model less general and opportunities for reuse could be missed. Another possibility is to not model a construct at all. This typically allows to get rid of language specifics, but also makes the model less useful.

In our view the chosen mappings, most notably those of Java constructors to methods and Java interfaces and Smalltalk metaclasses, have worked out well. We especially found both Java mappings to easily fit and allow to exploit the similarities with other constructs. For the metaclass mapping the advantages are less clear. Method and Attribute refactorings can be applied to (members of) metaclasses without any problems, but the class refactorings are not applicable at all. An alternative would be to not model metaclasses explicitly and model metaclass methods and attributes as class (in Java static) methods and attributes of the class the metaclass is representing. We have chosen not to do this, because, as said, some refactorings do work with this scheme and the alternative mapping results in problems with name clashes between class methods and instance methods and problems with the equal treatment of instance level class attributes and class level instance attributes which are different concepts in Smalltalk.

A last word about supporting other languages than the ones discussed in this paper. The FAMIX model is already set up to support more languages than Smalltalk and Java. For reverse engineering purposes we have used the model for C++ and Ada as well [16][15]. Therefore, we are confident we can use our model and extend our tool to support these and other languages without too many problems.

8. Conclusion

We have presented the results of a feasibility study concerning refactorings for multiple object-oriented languages. The main conclusions of our experiment are:

- A meta-model including concepts to represent classes, methods, attributes, inheritance, method invocations and attribute accesses is a necessary and sufficient basis to test the preconditions for the majority of the primitive refactoring operations. A subset of this meta-model would be insufficient to test all preconditions, while a richer one would support more refactorings but would become too language-dependent.
- Based on this meta-model, it is possible to construct a refactoring engine that performs primitive refactoring operations for a representative pair of implementation languages, namely Smalltalk and Java. Such a refactoring engine necessarily includes a language-dependent part, but this part can be kept sufficiently small to show that a language independent refactoring engine is worthwhile.

Apart from increasing the understanding of refactorings and the differences between the supported languages, separating the analysis for refactorings in a language-independent and a language-dependent part has basically the advantage that complex analysis can be reused for many languages. This is particularly relevant for hybrid tool environments that need to support many languages, repository-based CASE tools being the most notable examples.

In the future we first of all plan to perform more experiments with our refactoring engine. The presented work is an initial feasibility study and needs more work to be conclusive. Especially we will be working with the tool on real world code as opposed to a toy case study. Furthermore we plan to extend the number of supported refactorings and cover more languages such as C++, Ada and possibly procedural languages such as COBOL. Apart from that we will be exploring the combination of refactoring with program analysis techniques, aiming at language-independent analysis tools that propose improvements in terms of refactorings.

Acknowledgements

This work has been funded by the Swiss Government under projects NFS-2000-46947.96, BBW-96.0015 and BBW 00.0170 as well as by the European Union under the ESPRIT programme Project no. 21975 (FAMOOS) and IST-1999-20398 (PECOS). Furthermore, we thank Matthias Rieger for his useful comments on earlier drafts of this paper.

References

[1] E. Casais, "An Incremental Class Reorganization Approach," *Proceedings ECOOP'92*, O. Lehrmann Madsen (Ed.), LNCS 615, Springer-Verlag, June 1992, pp. 114-132.

[2] S. Demeyer, S. Ducasse and S. Tichelaar, "Why Unified is not Universal. UML Shortcomings for Coping with Round-trip Engineering," *Proceedings UML'99*, B. Rumpe (Ed.), LNCS 1723, Springer-Verlag, October 1999.

[3] S. Demeyer, S. Tichelaar and P. Steyaert, "FAMIX 2.0 - The FAMOOS Information Exchange Model," Technical Report, University of Berne, August 1999.

[4] S. Ducasse, M. Lanza and S. Tichelaar, "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems," *Proceedings of CoSET 2000*, June 2000.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[6] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. thesis, University of Illinois, 1992.

[7] D. Roberts, J. Brant and R. E. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems (TAPOS)*, vol. 3, no. 4, 1997, pp. 253-263.

[8] S. Tichelaar, "FAMIX Java language plug-in 1.0," Technical Report, University of Berne, September 1999.

[9] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[10] D. B. Roberts, "Practical Analysis for Refactoring," Ph.D. thesis, University of Illinois, 1999.

[11] M. M. Werner, "Facilitating Schema Evolution With Automatic Program Transformation," Ph.D. thesis, Northeastern University, 1999.

[12] M. Ó Cinnéide and P. Nixon, "A Methodology for the Automated Introduction of Design Patterns," *Proceedings ICSM'99*, 1999.

[13] L. Tokuda and D. Batory, "Automating Three Modes of Evolution for Object-Oriented Software Architecture," *Proceedings COOTS'99*, 1999.

[14] S. Ducasse and S. Demeyer (Eds.), *The FAMOOS Object-Oriented Reengineering Handbook*, University of Berne, October 1999, See http://www.iam.unibe.ch/~famoos/handbook.

[15] R. Nebbe, "FAMIX Ada language plug-in 2.2," Technical Report, University of Berne, August 1999.

[16] H. Bär, "FAMIX C++ language plug-in 1.0," Technical Report, University of Berne, September 1999.

[17] T. C. Lethbridge, "Requirements and Proposal for a Software Information Exchange Format (SIEF) Standard," Technical Report, University of Ottawa, November 1998, http://www.site.uottawa.ca/~tcl/papers/sief/standardProposalv1.html.

[18] Object Management Group, *Unified Modeling Language (version 1.3)*, Object Management Group, June 1999.