Dynamic Refactorings: improving the program structure at runtime

Peter Ebraert[*], Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussel, Belgium
{pebraert, tjdhondt}@vub.ac.be

Abstract

Many software systems must always stay operational, and cannot be shutdown in order to adapt them to new requirements. For such systems, dynamic software evolution techniques are needed. In this paper, we show how a restructuring technique – called refactoring – can be applied on running systems in order to facilitate future evolutions. We extend the pre- and post-conditions of the basic refactorings in order to ensure application consistency before and after the restructuring takes place.

1 Introduction

People always say that you should never change a system that is working fine. However, even if a software system seems to work properly from a user's point of view, it may be difficult to maintain or adapt from a developer's point of view. As such, it may be very cumbersome to evolve the system by adding a new feature, fixing a bug or porting the system to a new environment.

In all these situations where a software system is not flexible enough to allow a certain change, the technique of *software refactoring* can be used. According to Fowler [1], a refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour". Refactorings can be used to simplify the structure of a software system in order to prepare it for a certain evolution step.

Now suppose we have a running system, and we would like to evolve it without shutting it down. This is a much bigger challenge since there are considerably more constraints on the running system. Refactoring techniques would be very useful here too. For example, by reducing the coupling between objects in a running system, we could at the same time increase the system performance (from a user point of view) and its understandibility and evolvability (from a developer point of view).

Until now, refactorings have only been investigated in the context of source code restructuring. The main contribution of this paper is to show the use and feasibility of applying *dynamic refactorings*, i.e., refactorings that modify a running system.

[*]Authors funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)"

2 Atomic Change Sequence

Refactoring object-oriented programs typically replaces a set of classes C_1, C_2, C_3, ... by their new versions C'_1, C'_2, C'_3,... We use the notation ΔC_i to denote the difference between C_i and C'_i. In this section, we first define a set of atomic changes that can be used as the building blocks for specifying the *atomic change sequence* – the ΔC that must be applied in an atomic way in order to apply the corresponding refactoring.

Most of the differences we want to express can be represented by methods and instance variables. This is why our meta-object protocol currently implements the set of atomic change transactions that is shown in table 1. In the future we intend to extend this set to cover a more realistic set of applications.

Scope	Atomic Change	Explanation
Class	AC	Add an empty Class.
	DC	Delete an empty Class.
	CC	Changes a Class name.
Variable	AV	Adds an instance variable to a class.
	DV	Remove an instance variable from a class.
Method	AM	Adds a method to a class.
	DM	Deletes a method from a class.
	CM	Changes the implementation of a method.
	ML	Change the Method Lookup.

Table 1. Atomic Changes

The most simple atomic changes incorporate the ones on the scope of classes: adding empty classes (AC), deleting empty classes (DC), and changing a class name (CC) have a small impact on the system and can be performed without too many constraints. The only pre-requisit of the AC and CC is that name-clashes should be avoided, and only be tolerated only when intended (in case of polymorphism).

The changes on the scope of instance variables are a bit harder. As a result of an added variable (AV), all objects that are instance of this class have a new variable they can use to store values. By default, the value will be set to `nil`. A deleted variable (DV), deletes the variable in all the instances of the class. This is a dangerous operation as it could lead to runtime exceptions, if the variable is still used somewhere. That is why this atomic change has a prerequisite which states that the variable is not used throughout the system. Note that there is no operation of modifying a variable, as that can easily be modeled by deleting and adding the variable.

Finally, there are the changes that affect the methods. As a result of an added method (AM), all objects that are instance of this class will automatically understand this new method thanks to the languages method lookup mechanism. When a delete method (DM) is applied, all instances of this class may no longer understand this method. Hence, one should be very careful with this operation as it can give rise to runtime exceptions. The same counts for a changed method (CM), as this also has an impact on all objects that are instance of this class or one of its subclasses. This is why a CM and a DM have the prerequisites that the method is either still visible (somewhere up the inheritance chain), or either not called anywhere in the system.

3 Proposed solution

The following section describes the extension we want to make to the already known refactorings so that they can be safely applied to running systems. We start by explaining two basic refactorings and show how they are characterized in [1]. We show that the mechanics of applying the refactorings – as they are presented by Fowler – are not sufficient for applying the refactorings in a safe way on a running system. We then introduce extra prerequisites which ensure safety, and exemplify them by means of the corresponding two dynamic refactorings. We conclude the section by showing how the dynamic refactorings can be carried out on a running system.

3.1 Static Refactorings

In [1], Fowler presents a catalog of frequent refactorings. Every refactoring consists of a name, a motivation and the mechanics for applying the refactoring. We chose to explain

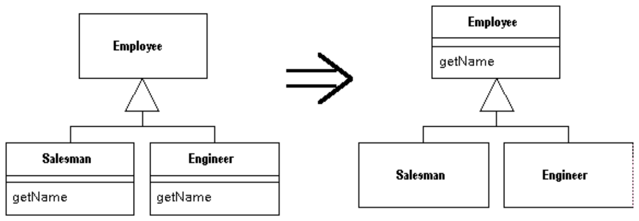

Figure 1. Pull Up Method.

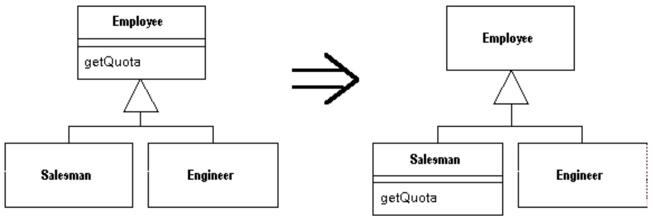

Figure 2. Push Down Method.

our approach on 2 refactorings: "Pull Up Method" ([1] page 322) and "Push Down Method" ([1] page 328).

3.1.1 Pull Up Method

Fowler introduced this refactoring for getting rid of unneeded code duplication. Its idea is to lift the common behavior of some classes to a common superclass as shown in figure 1. In table 2 we show the mechanics of the refactoring (represented as an atomic change sequence). As the pulled up method remains visible for all the subclasses and may even become visible for more classes, there can only be an addition of behavior. Every step has its pre- and post-conditions which must hold, for the refactoring to be valid. We can see that an AM leads to the *visibility* of that method in the class and its subclasses. We also see that before a DM, we must assure that the method is either not called anymore, or either visible in one of the superclasses. Those requirements are captured in the pre- and post-conditions of the atomic changes.

3.1.2 Push Down Method

Fowler introduced this refactoring for making the system behave in a more logical way, by specifying the behavior in the place where it makes more sense. Figure 2 shows that the refactoring is used for moving some behavior from a super class to a subclass. In table 3 we show the mechanics (represented as an atomic change sequence). Pushing down a method can be seen as a removal of behavior for all the classes in which the method is not visible anymore.

Place	Pre	Change	Parameters	Post
Employee		AM	”getName”	visible
Salesmen	no callers or still visible	DM	”getName”	
Engineer	no callers or still visible	DM	”getName”	

Table 2. The atomic change set for the Pull Up Method

Place	Pre	Change	Parameters	Post
Salesmen		AM	”getName”	visible
Engineer		AM	”getName”	visible
Employee	no callers or still visible	DM	”getName”	
Engineer	no callers or still visible	DM	”getName”	

Table 3. The atomic change set for the Push Down Method

3.2 Extra needs for dynamicity

The difference between stopped and running systems lies in the system state, which is only incorporated in the latter. In fact, a running system can be seen as moving from one consistent state to another while the processing of transactions goes on. A consistent state is a state from which the application can continue processing normally, without processing to an error state. When applying refactorings at runtime, we should always make sure that the application state remains consistent *before* and *after* the application of the update.

For ensuring state consistency before the application, we must make sure that the affected classes are in a *quiescent* status. An object in a quiescent status was proven to remain in a consistent state [2]. An object is in a quiescent status if: (i) it is not currently engaged in a transaction that it initiated, (ii) it will not initiate new transactions, (iii) it is not currently engaged in servicing a transaction, and (iv) no transactions have been or will be initiated by other objects which require service from this objects [2]. Theoretically, quiescence is achieved by adding extra preconditions which must hold before a refactoring can be applied. Practically, those preconditions are met by deactivating all objects that are affected by the refactoring. The deactivation and activation itself are added to the atomic change sequence of the refactoring. The post-condition of a deactivation is that the object is in a quiescent status $Q(O)$. The post-condition of an activation is that the object is in an active status $A(O)$.

Ensuring state consistency after the update clearly depends on the update itself. In our case, the updates only consist of refactorings. Since Fowler defined a refactoring as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour”, we are by definition only making structural changes. If we make those changes in a correct way, state consistency can be easily assured.

3.3 Dynamic Refactorings

Extending the atomic change sequence of a refactoring is a process that can be automated. We need to add preconditions, postconditions, and actions that make sure that those pre- and post-conditions are eventually met. In general, we first establish the set of all classes that are affected by the refactoring, and for each of them, we add a quiescence pre-condition. In practice, however, this process can be optimized, since quiescence is not needed for certain classes that are involved in the refactoring. For example adding a method to a class can never lead to run-time errors. We now exemplify the extension of the atomic change sequences by showing the two refactorings that were presented before.

3.3.1 Dynamic Pull Up Method

In table 4 we show the mechanics of the dynamic refactoring (represented as an atomic change sequence). We can see that the actual refactoring is performed when the affected classes reside in a quiescent status and that it is surrounded by actions that ensure quiescence.

3.3.2 Dynamic Push Down Method

In table 5 we show the mechanics of the dynamic refactoring (represented as an atomic change sequence). Again, we can see that the actual refactoring is performed when the affected classes reside in a quiescent status and that it is surrounded by actions that ensure quiescence.

Place	Pre	Atomic change	Parameters	Post
Salesmen	A(Salesmen)	Deactivate		Q(Salesmen)
Employee	A(Employee)	Deactivate		Q(Employee)
Engineer	A(Engineer)	Deactivate		Q(Engineer)
Employee		AM	”getName”	visible
Salesmen	Q(Salesmen), no callers or still visible	DM	”getName”	
Engineer	Q(Engineer), no callers or still visible	DM	”getName”	
Salesmen	Q(Salesmen)	Activate		A(Salesmen)
Employee	Q(Employee)	Activate		A(Employee)
Engineer	Q(Engineer)	Activate		A(Engineer)

Table 4. The atomic change set for the Dynamic Pull Up Method

Place	Pre	Atomic change	Parameters	Post
Salesmen	A(Salesmen)	Deactivate		Q(Salesmen)
Employee	A(Employee)	Deactivate		Q(Employee)
Engineer	A(Engineer)	Deactivate		Q(Engineer)
Salesmen		AM	”getName”	visible
Engineer		AM	”getName”	visible
Employee	Q(Employee), no callers or still visible	DM	”getName”	
Engineer	Q(Engineer), no callers or still visible	DM	”getName”	
Salesmen	Q(Salesmen)	Activate		A(Salesmen)
Employee	Q(Employee)	Activate		A(Employee)
Engineer	Q(Engineer)	Activate		A(Engineer)

Table 5. The atomic change set for the Dynamic Push Down Method

3.4 Carrying out the refactoring

From the moment we have the change transaction sequence that corresponds to a certain refactoring, we can start carrying out these changes on the running system. The changes are applied one by one on the running system, but in an atomic way (or all changes are applied, or none of them is applied). While most of the changes can be done transparently, some may require the programmer's interference. This is the case when there is a state involved, that needs to be preserved. Concretely, when an instance variable is deleted or modified, the question arises what has to happen with the value of that instance variable. Either the value can be ignored, or its is needed later in a new instance variable that will be added. Consequently, when an instance variable is added, the programmer is also requested to interfere, and to tell wether the variable should be initialized with a certain value. For example, using Euros instead of Belgian Francs in our bank accounts requires us to use the following formula: 'take the old value and multiply it by 40,3399, and use it as the new value'. For methods in class-based systems, things are much simpler. Because methods are only referenced through the class itself, adapting them on the class level does the job. Making these changes is done in practice by using interceptive techniques [3], which incorporate all the atomic changes that are specified in the meta-object protocol.

Ensuring quiescence is the hardest part, and consists of two phases. First, we need to find all the affected objects. In a class-based language, the set of affected objects of a change on class C consists of the class itself, its subclasses and all instances of those classes. Practically, this set can be assembled by using introspective techniques [3], which allow us to query a class for all its subclasses and instances.

In the second phase we have to ensure for each of the affected objects, that: (i) it is not currently engaged in a transaction that it initiated, (ii) it will not initiate new transactions, (iii) it is not currently engaged in servicing a transaction, and (iv) no transactions have been or will be initiated by other objects which require service from this objects. In class-based programming languages, (i) and (iii) can be assured by making sure that the object is not on the runtime stack. (ii) and (iv) can be assured by making sure that no messages will be send to the affected object.

4 Experimental Setup

Because the focus of this paper is on refactorings, we restrict ourselves to class-based object-oriented languages.

Object-oriented languages like Java are excluded because of the limitations of their reflective capabilities. Smalltalk, on the other hand, is fully reflective: everything is an object, and can thus be taken apart, queried for information and possibly be modified. Even messages are objects, and can thus be monitored and modified when they are sent or received [4]. This is why we chose Smalltalk as the language in which we plan to conduct the experiments.

The Smalltalk development environment is very dynamic, in the sense that developing smalltalk code happens in an incremental way, by inspecting the newly created classes and objects. This is why developing Smalltalk programs actually happens at runtime. The Smalltalk Refactoring Browser provides support for static refactorings. But because it does not check for the extra requirements that we have presented in this paper, it sometimes fails to ensure consistency. That is why we plan to extend the Smalltalk Refactoring Browser with the support for dynamic refactorings. We will make sure that, before a refactoring is performed, the refactoring browser will check for the additional requirements, making sure that the system remains in a consistent state.

5 Evaluation

The main benefit of applying refactorings at runtime is the preservation of the state and object identity, as we will keep on working on the same (already existing) class C. Replacing a class C would involve the creation of C', the swapping of all relations from C to C', the deletion of C and the mapping of the state from C to C'. Evolving the existing C component to C' only involves the creation of C' and the propagation of the changes on C. This implies that there will be no more relation swapping problems and less state mapping problems.

While it is clear that this approach is impossible without reflection, reflection itself also hinders the approach. Ensuring consistency on a program that uses reflection is a lot harder then on a program that does not allow reflection. So in a sense reflection can be seen as both our best friend, and our enemy.

In this position paper, we presented two static refactorings and their dynamic equivalents. It is our goal to present a dynamic equivalent for all of the refactorings that were identified in [1]. For doing so, we will have to extend our metaobject protocol and maybe introduce new prerequisites. This, however, is subject for further investigation.

6 Conclusion

In some cases, software systems can not be turned off for carrying out an evolution. This triggers the need for supporting dynamic evolution. We suggested to apply dynamic refactorings to improve the runtime component structure of object-oriented software systems for easing future evolution. The approach relies on the reflective properties of the underlying programming language in order to modify the application's behavior.

We start from the static refactorings that were presented by Fowler in [1], and extend their mechanics in such a way that they can be applied on a running system without braking consistency. For doing that, we use the quiescence property, that was introduced in [2] as a sufficient condition for consistency. We add quiescence as a prerequisite for the refactorings and show how this can be implemented in a class-based environment that has full reflective capabilities. We envision the extension of the Smalltalk Refactoring Browser for experimenting with the dynamic refactorings.

References

[1] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)

[2] Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering **16** (1990) 1293–1306

[3] Maes, P.: Computational Reflection. PhD thesis, Artificial Intelligence Laboratory, Vrije Universiteit Brussel (1987)

[4] Messick, S.L., Beck, K.L.: Active variables in smalltalk-80. In: Technical Report CR-85-09, Computer Research Lab, Tektronix (1985)