Assisting System Evolution: A Smalltalk Retrospective

Robert Hirschfeld[*] Matthias Wagner[*] Kris Gybels[#]

[*]DoCoMo Euro-Labs
[#]Vrije Universiteit Brussel

hirschfeld@docomolab-euro.com
wagner@docomolab-euro.com
kris.gybels@vub.ac.be

Abstract

This paper illustrates selected Smalltalk mechanisms supporting software system evolution. System modifications considered are the renaming of a class, the removal of a class, and the change of the layout of a class. The mechanisms discussed are in use in Smalltalk implementations for a long time and have proven to be effective for unanticipated software evolution. This paper gives a retrospective overview in this context.

1 Introduction

For critical applications, providing highly available services becomes increasingly important. High availability has to cope with modifications necessary to match changes in the system's environment, the requirements, and the understanding of the problem domain.

Dynamic software systems are designed to enable applications to be modified during development and at runtime. Accordingly, dynamic systems are capable of leveraging the minimization of allowable downtime. This paper shows how a dynamic system like Smalltalk can assist in dealing with the issues stated above. The Smalltalk environment referred in this text is Squeak, an open highly portable Smalltalk-80 implementation with a virtual machine written entirely in Smalltalk [Sque01, GoRo83].

2 Selected Evolution Scenarios

Dynamic programming in Smalltalk has a long history. Due to late binding and metaprogramming, Smalltalk allows both behavior and structure to be evolved at runtime.

This paper illustrates Smalltalk's approach to deal with system evolution. Discussed are the following selected scenarios:

- the renaming of a class,
- the removal of a class, and
- the change of the layout of a class.

Renaming a class requires the maintenance of references to the renamed class that is typically referred to via a lookup by its name in the system dictionary. Removing a class might leave instances created earlier in the system. Such instances have to be

kept operational for a transitional period of time in order to preserve system consistency. Changing the layout of a class can lead to a situation where instances conforming to the previous layout either have to stay active, or have to be transformed to fit the new layout.

3 Renaming a Class

The system dictionary (referred to by the global variable Smalltalk) holds associations that relate class names to class objects. A class can be accessed by its name via an explicit lookup in the system dictionary, directly via a reference to the class object, or indirectly via a reference to the association in the system dictionary.

Compiled methods take advantage of the latter technique when they have to refer to classes (or global variables in general) in their literals section.

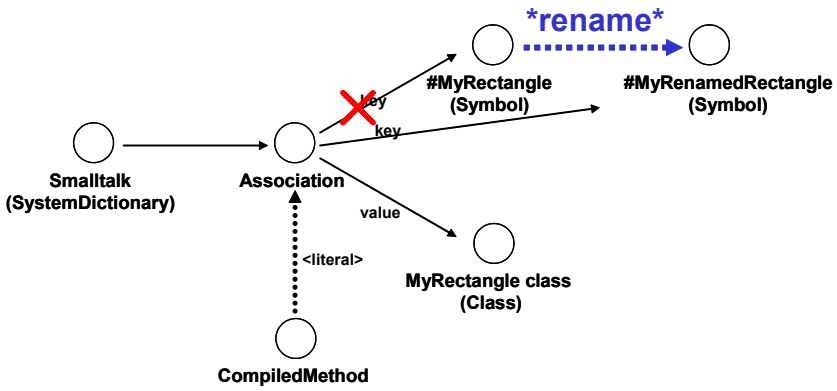

Figure 1: Renaming a Class

Figure 1 illustrates structural changes involved in renaming a class: here, a class object named #MyRectangle is renamed to #MyRenamedRectangle. The graphical notation used in this text is based on OOSE/Objectory [JCJÖ93]. In general, most classes are placed in the system dictionary via an association referring to both the name of the class, and the class object itself. This indirect reference is of advantage because renaming only affects the key part of the association holding the class name, whereas the value part of the association holding the class object itself stays the same.

This is achieved by explicitly keeping the association object, removing the key of the association representing the old name of the class (#MyRectangle) from the system dictionary, changing the key of this association to become the new name of the class (#MyRenamedRectangle), and then reinserting the original association into the system dictionary.

4 Removing a Class

Removing a class might leave instances in the system that were created prior to the removal of their class and that are still referenced by other objects. Such instances have to be kept operational for a transitional period of time in order to preserve system consistency and to prevent failure situations.

When a class is removed, it is taken out of the system dictionary, but is kept in the system and marked as obsolete as long as there are instances of it. The whole structure behind a class covering compiled methods and their references to the association referring back to the actual class object itself is preserved. With that, messages understood before can still be received by respective instances, and instances of an obsolete class can be created from methods that, at compile time, referred to the now obsolete class by name. Note that the Smalltalk system supports developers in finding references to obsolete classes. Also, Smalltalk does not allow new methods to be compiled with references to obsolete classes or to access them by their name.

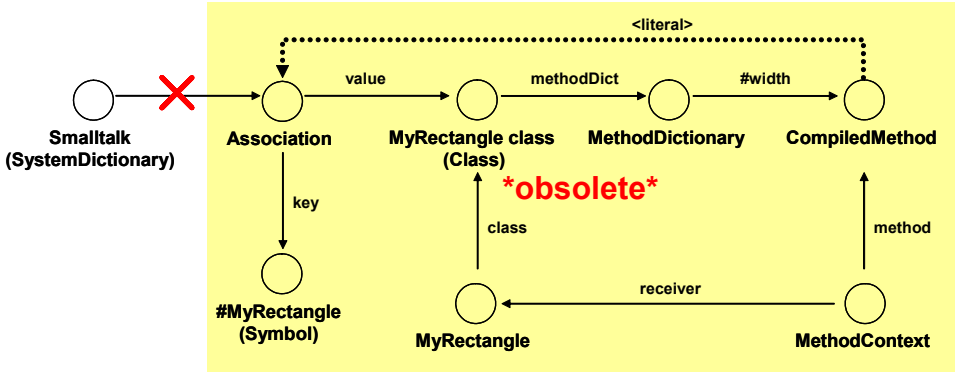

Figure 2: Removing a Class

Figure 2 shows the removal of class MyRectangle: the association relating the class with its name is removed from the system dictionary, but kept intact for compiled methods referring to that class object indirectly via this association.

5 Changing the Layout of a Class

While behavior inheritance is dynamic (done every time an object receives a message), structure inheritance is static (the format of an instance of a class is computed once at the definition or redefinition of a class) [Paep93]. Changing the behavioral part of the system is considerably straightforward, whereas changing certain structural aspects is more complex.

In many Smalltalk implementations, objects are stored on the heap. When a new object is created, space is obtained to store the object header (stating the size of the object and the class it belongs to) and all its fields (representing instance variables) into an adjacent sequence of memory words.

In the source code of a method, the access to instance and class instance variables is expressed via the variables name. The Smalltalk compiler translates the source code of a method into byte code, a sequence of instructions for the Smalltalk interpreter, stored in compiled methods. In the byte code, instance variables are accessed via an integer index into the sequence of adjacent fields, representing instance with pointers to other objects.

The layout of a class depends on the number of instance variables and their order. The addition, removal, or rearrangement of instance variables, or the re-parenting of a class affect a class' layout. As a consequence, compiled methods with byte code

referencing instance variables are left with invalid indices and therefore have to be recompiled.

The result of the recompilation after a layout change are compiled methods with proper indices for referencing instance variables of objects created with the new layout. However, these new compiled methods won't work with existing instances created with an older layout. Respective actions have to be taken to keep the system in a consistent state.

The following example will illustrate this fact (Figure 3): Instances of class MyRectangle have two instance variables named origin and corner. Their #width method answers the width of an instance. The original byte code sequence for #width shows that corner is accessed via the zero-relative index of 1, and origin via the zero-relative index of 0.

Method accessing named instance variables:

```
MyRectangle>>width
        "Answer the width of the receiver."
        ^ corner x - origin x
```

Original shape (MyRectangle):	**New shape (MyRectangle'):**
Object subclass: #MyRectangle instanceVariableNames: 'origin corner ' classVariableNames: '' poolDictionaries: '' category: 'Obsolete-Test'	Object subclass: #MyRectangle instanceVariableNames: 'extent origin corner ' classVariableNames: '' poolDictionaries: '' category: ' Obsolete-Test'
Original byte code (MyRectangle>> width):	**New byte code (MyRectangle'>>width'):**
5 <01> pushRcvr: 1 6 <CE> send: x 7 <00> pushRcvr: 0 8 <CE> send: x 9 <B1> send: - 10 <7C> returnTop	5 <02> pushRcvr: 2 6 <CE> send: x 7 <01> pushRcvr: 1 8 <CE> send: x 9 <B1> send: - 10 <7C> returnTop

Figure 3: Changing the Layout of a Class (Instance Variables indexed from Byte Code)

After changing the shape of MyRectangle (into MyRectangle') by inserting a new instance variable, in the example at the beginning of the instance variable sequence, a recompilation of #width results in a new #width' with a different byte code sequence, now accessing corner via the zero-relative index of 2 and origin via the zero-relative index of 1. This shows that the new compiled method won't work on instances with the original layout and vice-versa.

Squeak handles this dilemma as follows (Figure 4): A new class object is created conforming to the new layout (the class format) and pointing to the same objects as the old class object. All methods of the old class, working with the old layout, are recompiled in the context of the new class to conform to the new layout. For each instance of the old class a new instance of the new class is created, with matching

instance variables pointing to the same objects. After this adjustment of references to other objects, references from other objects to the old instances are redirected to now point to respective new instances. Finally, all pointers to the old class are corrected to point to the new class instead.

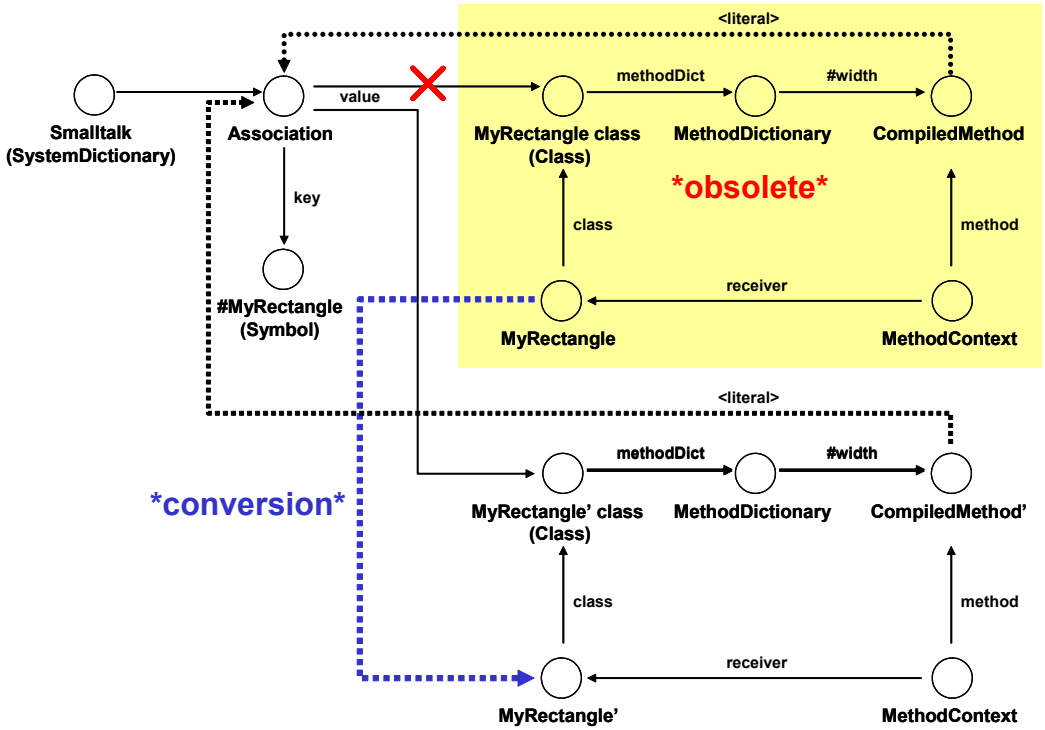

Figure 4: Changing the Layout of a Class

The whole procedure is executed in a setup that prevents the possibility of preemption by another process with higher priority. As a result, the old class structure was rendered obsolete, and all of its instances were morphed into instances of the new class, reshaped to conform to the new layout.

6 Conclusion

Highly available systems, especially in the area of telecommunications, are expected to minimize allowable down time. Supporting unanticipated changes and run-time evolution in systems like these is critical. Avoiding extensive invasive modifications of existing code is becoming a major issue in software engineering. Dynamic development and run-time environments are able to cope with complex aspects of system evolution. There are ongoing research activities to introduce selected mechanism of dynamic programming languages and software systems into more static environments [CStC01, Dmit01, Knie99].

This paper studies the way Smalltalk is dealing with selected evolution scenarios. Systems to be enabled for software evolution will significantly benefit from dynamic environments supporting late binding and meta programming. There is a lot to be learned from environments like Smalltalk that were designed to support evolving systems.

Acknowledgements

Thanks are due to David Simmons and Peter Schoo for their helpful comments and suggestions.

References

[GoRo83] Goldberg, Adele; Robson, David:
Smalltalk-80: The Language and Its Implementation.
Addison-Wesley, 1983

[CStC01] Costanza, Pascal; Stiemerling, Oliver; Cremers, Armin B.:
Object Identity and Dynamic Recomposition of Components.
In: Proceedings of TOOLS Europe, Zurich, 2001

[Dmit01] Dmitriev, Mikhail:
Safe Class and Data Evolution in Large and Long-Lived Java Applications.
Sun Microsystems Laboratories, TR-2001-98, August 2001

[JCJÖ93] Jacobson, Ivar; Christerson, Magnus; Jonsson, Patrik; Övergaard, Gunnar:
Object-Oriented Software Engineering – A Use Case Driven Approach.
Addison-Wesley, 1993

[Knie99] Kniesel, Günter:
Type-Safe Delegation for Run-Time Component Adaptation.
In: Proceedings of ECOOP99, Springer LNCS 1628, 1999

[Paep93] Paepcke, Andreas:
Object-Oriented Programming – The CLOS Perspective.
The MIT Press, 1993

[Sque01] *Squeak homepage.*
(http://squeak.org)