Lessons in Software Evolution Learned by Listening to Smalltalk *

Oscar Nierstrasz and Tudor Gîrba

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch

Abstract. The biggest challenge facing software developers today is how to gracefully evolve complex software systems in the face of changing requirements. We clearly need software systems to be more dynamic, compositional and model-centric, but instead we continue to build systems that are static, baroque and inflexible. How can we better build *change-enabled* systems in the future? To answer this question, we propose to look back to one of the most successful systems to support change, namely Smalltalk. We briefly introduce Smalltalk with a few simple examples, and draw some lessons for software evolution. Smalltalk's simplicity, its reflective design, and its highly dynamic nature all go a long way towards enabling change in Smalltalk applications. We then illustrate how these lessons work in practice by reviewing a number of research projects that support software evolution by exploiting Smalltalk's design. We conclude by summarizing open issues and challenges for change-enabled systems of the future.

1 Introduction

The conventional view of disciplined software construction is to reason that *correctness* of the final result is paramount, so we must invest carefully in rigorous requirements collection, specification, verification and validation.

Of course these things are important, but the fallacy is to suppose that there *is* a final result. This leads one to the flawed corollary that it is possible to get the requirements right. The truth (as we know) is that in practice *evolution* is paramount [26,32], so the system is never finished, and neither are its requirements [4].

What features are important in a software system to enable graceful software evolution? In previous work we have argued that evolution is enabled by high-level composition of components [2]. We have also argued that such systems should also be dynamic, they should support reflection on-demand, and they should provide mechanisms to manage the scope of change [33]. Change should be represented as a first-class entity, and both static and dynamic models of the running applications should be available at run-time to support continuous monitoring and analysis of evolution [34]. Instead of being merely "model-driven",

* J. van Leeuwen et al. (Eds.): SOFSEM 2010, LNCS 5901, pp. 77–95, 2009. Invited paper. © Springer-Verlag Berlin Heidelberg 2009.

such systems should be *model-centric*, meaning that models are not only available for analysis, but also to enable and enact change. To control the scope of change, systems need to be *context-aware*, thus allowing selected changes to be visible to different parts of the same running system [35]. In a nutshell, change-enabled systems should be (i) compositional, (ii) dynamic, (iii) model-centric, (iv) reflective, (v) self-monitoring, and (vi) context-aware.

But how should we build such change-enabled systems? What are good examples of systems that actively support and enable rather than limit and impede software evolution?

In this paper we take the position that many of these questions can be partially answered by taking a close look at the Smalltalk system. Smalltalk [19,23] was the first programming language and development environment designed to be fully object-oriented from the ground up[1]. Many technical and process innovations arose from Smalltalk, including the first interactive development environments with graphical user interfaces, many virtual machine advances, refactoring tools, unit testing frameworks, and so on. Although it shows its age today, in many ways Smalltalk (like ALGOL [21]) still improves on its successors.

Smalltalk is still interesting today because it offers many features that support graceful software evolution. First of all, at the core it is very simple. Smalltalk is built up from a small set of fully object-oriented principles, starting with the notions that *everything is an object* and *everything happens by sending messages*. The syntax is remarkably simple, and can be read aloud like pidgin English. Second, it is fully reflective, so all features of the Smalltalk system are available at run-time as ... objects. Third, Smalltalk is highly dynamic. While most programming languages are trapped in an edit/compile/run cycle, Smalltalk supports incremental and interactive development of *running applications* by erasing the artificial distinction between "compile-time" and "run-time".

In Section 2 we introduce Smalltalk by means of series of simple examples, and we draw three lessons that illustrate how Smalltalk support software evolution. In Section 3, Section 4, and Section 5 we review a series of research projects that demonstrate how Smalltalk's simplicity, its reflective design, and its dynamic nature enable change. In Section 6 we discuss several shortcomings of Smalltalk and open challenges for change-enabled systems of the future. We conclude with a few closing remarks in Section 7.

2 What Can We Learn From Smalltalk?

Smalltalk was designed to be the programming language and operating system for implementing a new generation of lightweight, interactive computers known as the Dynabook [22,23] (now recognizable as a precursor of today's laptops; see Figure 1). To build such a radically different kind of computer, Kay reasoned

[1] Simula-67 [6] was earlier, but essentially extended ALGOL with object-oriented constructs, rather than being fully object-oriented.

that the underlying language and system should be object-oriented from the ground up.

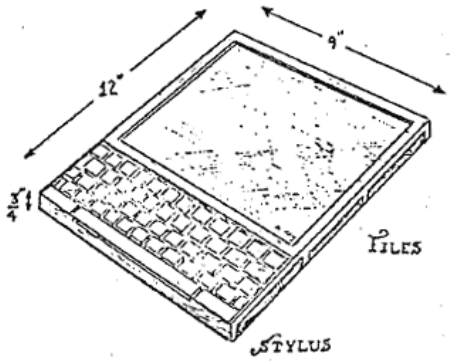

Fig. 1. Dynabook sketch from Kay's 1972 paper [22].

The principle "Everything is an object" pervades Smalltalk's design [19]. As we shall see, this simple starting point inevitably led to a design in which all aspects of Smalltalk are reified and available at run-time.

In this section we introduce Smalltalk through a series of simple examples that illustrate surprising aspects of Smalltalk's design principles. We conclude by drawing three lessons for designing change-enabled software systems.

2.1 Simple, Read-Aloud Syntax

Smalltalk as a language is pretty much minimal. It is common to remark that Smalltalk syntax can be learned in an afternoon, while the system itself can take many months to master.

Smalltalk supports three kinds of message syntax, as seen in the following example:

```
2 raisedTo: 1 + 3 factorial    ⟶    128
```

Unary messages, like factorial or new, consist of simple alphabetic identifiers, and are evaluated first. *Binary* messages, like +, are built up of operator symbols (much like in C++), always take a single argument, and are evaluated next. Finally, *keyword* messages, like raisedTo: or ifTrue:ifFalse:, consist of any number of keywords, each of which ends in a colon (:) and takes a single argument.

By exercising some common sense when naming classes, instance variables and methods, this scheme leads to compact code which can be read aloud as though it were a kind of pidgin English.

As a trivial example, try to read the following two roughly equivalent code fragments out loud:

```
for(int n=1; n<=10; n++){
   System.out.println(n);
}
```

```
1 to: 10 do: [:n | Transcript show: n; cr ]
```

By avoiding the need for most declarations, and by adhering to a message syntax that allows verbs and nouns to conveniently alternate, Smalltalk achieves a high level of readability. This is of course important if code is to be largely self-documenting. A large part of continuing development of complex software systems is *reading* of existing code, not just writing of new code.

2.2 Everything Happens by Sending Messages to Objects

Not only the syntax is simple, but also the design and implementation of Smalltalk follow logically from a few basic principles[19]. The most fundamental of these principles are:

1. *Everything is an object.*
2. *Everything happens by sending messages.*

Other important principles state, for instance, that:

3. *Every object is an instance of a class.*
4. *Every class (except the root) has a superclass.*
5. *Method lookup follows the superclass hierarchy.*

Everything is an object, including numbers, so when we compute 3 + 4, we send the message + to the object 3 with argument 4:[2]

```
3 + 4   ⟶   7
```

Both little numbers and very big numbers are objects:

```
42 factorial   ⟶
    1405006117752879898543142606244511569936384000000000
```

Since everything is an object, classes and methods are objects too. And since everything happens by sending messages, instantiating objects, defining new classes, and creating methods also happen by sending messages. For example, to define a class, we send a message to its superclass:

```
Object subclass: #Life
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MyUniverse'
```

[2] To indicate the result of evaluating a Smalltalk expression we use the notation
expression ⟶ result

We have just asked the root class Object to create a subclass of itself called Life in the category (read "package") called 'MyUniverse'.

To create a new object, we send a message to a class:

```
myLife := Life new
```

To define a method, we can send a message to its class:

```
Life compile: 'answer ↑ 42'
```

This method will be evaluated in response to the message answer, and returns[3] the result 42. Normally, however, we would define methods using the development environment, but it is important to remember that everything a tool does actually happens by sending messages.

Of course we can send messages to plain objects too:

```
myLife answer  ⟶  42
```

(You might have noticed that we just extended the behaviour of a living object.)

What is now more interesting is that we can now easily navigate and query the system as well simply by sending messages:

```
Life superclass              ⟶  Object
Life methods size            ⟶  1
Life methods first selector  ⟶  #answer
Life methods first class     ⟶  CompiledMethod
```

In this way we can quickly reach the meta-objects that implement the system (such as CompiledMethod). We can also easily explore the system's meta-model:

```
Life class              ⟶  Life class
Life class class        ⟶  Metaclass
Life class class class  ⟶  Metaclass class
```

This tells us that Life is an instance of Life class, that Life class is an instance of Metaclass, and that Metaclass is an instance of Metaclass class.[4]

2.3 Everything is There, All the Time

In Smalltalk, there is no distinction between the development environment and the runtime environment. They are one and the same.

The fact that all objects of the run-time system are accessible from the running image, and that all the source code is available all the time, leads to a very different style of development from the traditional file-based edit/compile/run life-cycle. Instead, Smalltalk encourages iterative and incremental development in which a single class is created or a single method is compiled at a time. We can change or extend the behaviour of already existing objects (*e.g.,* myLife acquires the answer method at run-time).

[3] ↑ is Smalltalk for "return".

[4] The alert reader might be able to conclude how this tale continues.

As a consequence, Test-Driven Development, in which failing tests are written before the code that makes the test pass, is naturally supported [5]. The surprising fact is that it is possible to add the missing code, using the Smalltalk debugger, *from the context of the failing test.*

Perhaps even more surprising is the extent to which it is considered best practice in Smalltalk to make heavy use of the debugger. By contrast, in most programming languages, the debugger is often considered a tool of last resort. In Smalltalk, since the entire system is live, the debugger provides a convenient interface to link source code to live objects. In other words, *the debugger is your friend.* Since code can be evaluated and even changed in the debugger, this leads to an interactive and incremental style of development in which one can modify and test a running application in very tight iterations.

Suppose, for instance, that we try to evaluate the following:

```
myLife meaning
```

Since our Smalltalk environment has never heard of the message "meaning", it asks us to confirm that this is what we intend. When we confirm, the object myLife receives the message meaning, but does not know what to do with it. This causes Smalltalk to send it the message doesNotUnderstand: with the symbol #meaning as its argument. The default behaviour is to launch a pre-debugger window which offers us the possibility of creating the missing method (Figure 2).

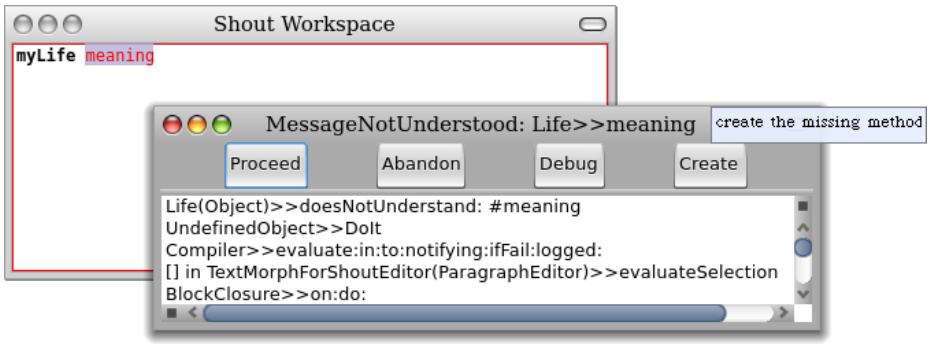

Fig. 2. Not understanding the meaning of life.

Smalltalk kindly generates a default implementation within the debugger, which does nothing but send the message shouldBeImplemented to self. From within the debugger we can change this method to something more reasonable (Figure 3).

Now if we ask Smalltalk to *Proceed*, we obtain the result we expect, *without ever having left the running system.*

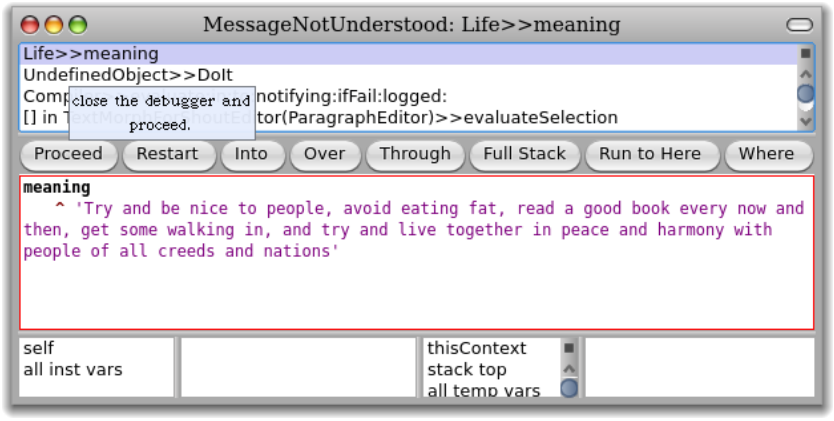

Fig. 3. Redefining the meaning of life.

```
myLife meaning  ⟶  'Try and be nice to people, avoid eating fat, read a good
                   book every now and then, get some walking in, and try and live together in
                   peace and harmony with people of all creeds and nations'
```

2.4 Lessons in Software Evolution

We have seen how Smalltalk distinguishes itself by its simplicity, its reflective design, and its dynamic nature. These features support software evolution in important ways:

- *Less is more*: Both the model and the syntax of Smalltalk are minimal. The model is extended by introducing new objects, not by changing the language. This minimal syntax allows for fluent interfaces to arise more naturally in Smalltalk than many other languages, thus the code is largely self-documenting — a critical feature for an evolving system.
- *Reify everything*: The design of Smalltalk follows logically from a small set of principles. This makes the system easy to navigate, query and extend.
- *You can change a running system*: Contrary to most other software systems, in Smalltalk you can *only* change a running system. There is no distinction between edit-time, compile-time and run-time. The entire Smalltalk system is described in itself. Essentially all the source code and the entire run-time system is accessible all the time. This makes it a good basis for realizing run-time, model-driven systems.

In the following sections, we will explore these points by reviewing several research projects that exploit Smalltalk to enable change.

Smalltalk also has quite a few wrinkles, grey hairs and creaky joints. For instance, Smalltalk's traditional support for modularity based on "categories" of

related classes is primitive at best. We will conclude this paper with a discussion of a number of areas where Smalltalk, and other programming systems, need to better address the needs of software evolution.

3 Less is More

The simplicity of Smalltalk's syntax makes it easy to learn. But, there is another important aspect that this simple syntax supports well, which is the design of *fluent interfaces* for black-box, component frameworks. A fluent interface resembles a domain specific language (DSL), except that it is entirely embedded in a host language, without requiring any syntactic extensions [18]. Fluent interfaces arise naturally with black-box frameworks, in which applications are built by plugging together existing components, as opposed to white-box frameworks, where applications are built by subclassing framework classes and implementing hook methods [43].

By carefully designing the interface of a black-box framework, compositions of components resemble readable (or "fluent") high-level "scripts" in a DSL. DSLs enable change by raising the level of abstraction, and by offering a more suitable notation for domain experts to express requirements. Let us review a number of examples.

Seaside. Consider the following example from an on-line store programmed using Seaside [15], a web application development framework written in Smalltalk:

```
renderContentOn: html
    html heading: item title.
    html heading level3; with: item subtitle.
    html paragraph: item description.
    html emphasis: item price printStringAsCents.
    html form: [
        html submitButton callback: [self addToCart]; text: 'Add To Cart'.
        html space.
        html submitButton callback: [self answer]; text: 'Done' ]
```

A reader who knows neither the specific application nor Smalltalk, but is familiar with HTML, should be able to read this aloud and make sense of it. The code reads like a script in a DSL, but is actually plain Smalltalk code using Seaside's fluent interface. The result can be seen in the *California Roll* item in Figure 4.

Mondrian. Mondrian [31] is a black-box framework for generating visualizations. The following script generates a simple System Complexity View [25] of a class hierarchy, mapping dimensions and shading of boxes to metrics (see Figure 5):

```
view := ViewRenderer new.
view nodeShape rectangle
    width: #NOA; height: #NOM;
    linearColor: #LOC within: model classes.
view nodes: model classes.
```

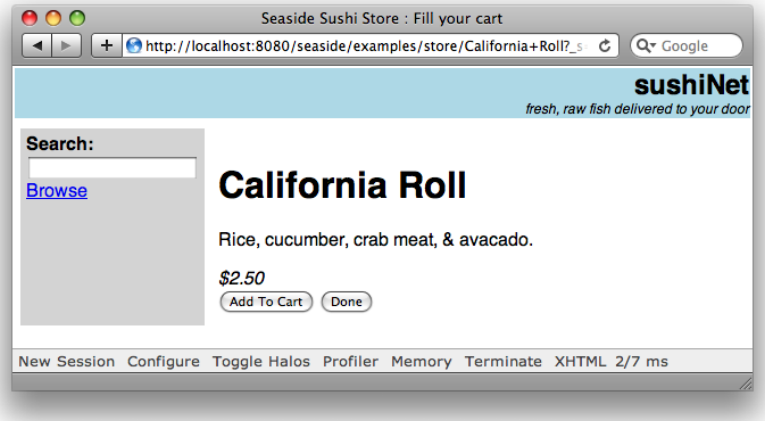

Fig. 4. Scripting a Seaside component.

```
view edges: model inheritances from: #superclass to: #subclass.
view treeLayout.
view open.
```

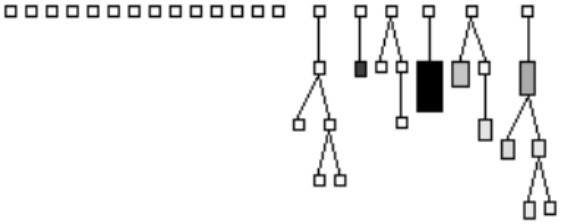

Fig. 5. A Mondrian-scripted System Complexity View

Glamour. Glamour is yet another black-box framework used to develop interactive browsers for diverse information models [8]. As with Seaside and Mondrian, Glamour scripts are compact, readable, and resemble code written in a dedicated DSL, though in fact they simply make use of a fluent interface written in Smalltalk. For example, the following script produces a file browser similar to Windows Explorer (see Figure 6).

```
browser := TableLayoutBrowser new.
browser
    column: #folders;
    column: [ :col | col row: #files span: 2; row: #preview ] span: 2.
```

```
browser showOn: #folders; using: [
    browser tree children: [ :folder | folder files select: #isDirectory ] ].
browser showOn: #files; from: #folders; using: [
    browser list display: [ :folder | folder files reject: #isDirectory ] ].
browser showOn: #preview; from: #files; using: [
    browser text display: #contentsOfEntireFile ] ].
```

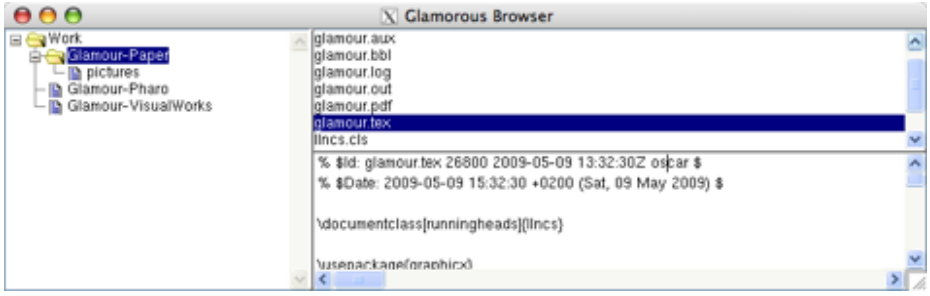

Fig. 6. A Windows Explorer-like browser implemented in Glamour.

Black-box frameworks separate what is *stable* (*i.e.,* the components) from what needs to stay *flexible* (*i.e.,* the scripts) [2]. High-level scripts facilitate software evolution by concentrating the *composition* of the system in a readable specification. A fluent interface makes scripts easy to read, and hence easy to modify and test, in contrast to traditional white-box frameworks which can be notoriously difficult to understand, specialize, and configure. Smalltalk's simple syntax and semantics supports both the development of pluggable black-box components and fluent interfaces to compose them.

4 Reify Everything

Smalltalk relies on a simple and explicit meta-model. With a simple meta-model, not only can we easily query our software, we can also extend it. In this section we will review a number of projects that have extended Smalltalk's meta-model to support software evolution.

Traits. Traits [16] extend Smalltalk's meta-model with reusable groups of methods, thus overcoming the limitations of single inheritance, while avoiding fragility problems known to occur with multiple inheritance and mixins. The introduction of traits required changes to the meta-model, the compiler, and the run-time, but no changes to the language or the syntax, since everything happens by sending messages.

Let us define a new trait:

```
Trait named: #TUltimate uses: {} category: 'MyUniverse'.
TUltimate compile: 'question ↑ ''What do you get if you multiply six by nine?'''
```

We change our class to use this trait:

```
Object subclass: #Life
    uses: TUltimate
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'MyUniverse'
```

And now:

```
myLife question    ⟶    'What do you get if you multiply six by nine?'
```

Traits support evolution by simplifying the refactoring of complex hierarchies into finer grained, reusable components [10].

Magritte. Model-driven engineering (MDE) promotes software evolution by raising the level of continuous development to levels that are closer to the problem domain. But conventional MDE makes use of transformations to *generate* platform-specific models (and code) from platform-independent models. The models are not typically available to the run-time system, so further adaptation and evolution are not possible at run-time.

A model-centric system [35] makes high-level, causally-connected models available to the run-time system for analysis and run-time adaptation. Smalltalk offers a good foundation for model-centric systems due to its reflective architecture.

A good example of a model-centric system is *Magritte* [38] a meta-description framework implemented in Smalltalk. Magritte has been used, for example, to meta-describe components of the Pier content management system[5], allowing it to be customized at run-time. Not only users can customize the domain model at run-time, but developers can directly customize many aspects of the meta-model without writing a line of code, since the meta-model is also meta-described and rendered by Pier as web components (Figure 7).

Moose. Moose[6] provides another example of a model-centric system. It offers a platform for capturing, querying, navigating, analyzing and visualizing models of complex software systems [36]. Several analyses have been built on top of it dealing with various aspects of software: static analysis, dynamic analysis, evolution analysis, semantic analysis, code duplication, code ownership analysis and so on.

These analyses require various meta-models. To accommodate them, at its core, Moose has a meta-meta-model in terms of which the various meta-models are defined [14,24]. Based on these descriptions, Moose offers import-export capabilities, it generates user interfaces for navigation, and provides integration mechanisms for extension services that encode specific analyses.

[5] `http://www.piercms.com`
[6] `http://moose.unibe.ch`

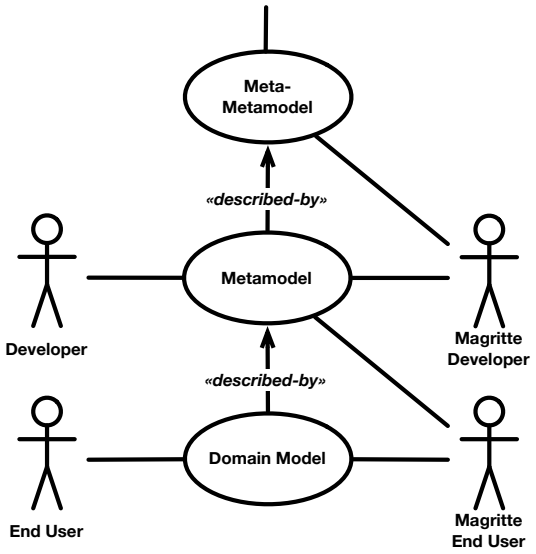

Fig. 7. Magritte enables run-time customization of models and meta-models.

5 You can Change a Running System

Since in Smalltalk, everything is an object, it follows that the Smalltalk system itself consists of a collection of objects. Since everything happens by sending messages, it follows that all changes to the system are simply consequences of messages being sent. In other words *changes to the system occur within the system*, and are no different than any other events.

The fact that everything is there all the time and can be changed dynamically means that both past and future evolution are accessible to the running system. We will briefly look at three ways this can be exploited.

Object-Flow Analysis. One of the well-known shortcomings of conventional stack-oriented debuggers is that the offending context which may have led to a run-time error may no longer be on the stack. If a method has left an object in an invalid state, this might produce an undesirable side effect at a much later point in time. A so-called *back-in-time debugger* [29] keeps track of historical execution contexts to allow the developer to debug further back in time. Although appealing, tracking history may generate vast amounts of data, and still it may be difficult to track the actual cause of a defect.

The *object-flow VM* [28] tracks history in a live Smalltalk system by *tracking the flow of objects* with first-class aliases, each of which stores a past state and records the previous alias which led to it. Since aliases are first-class, unreachable aliases are automatically garbage-collected, leading to a simple and elegant saving of space. Since aliases are managed at the VM level, they are invisible to running applications. *Compass* [27] is a back-in-time debugger implemented

using the object-flow VM, which exploits object flow to simplify navigation of the tree of past contexts (see Figure 8).

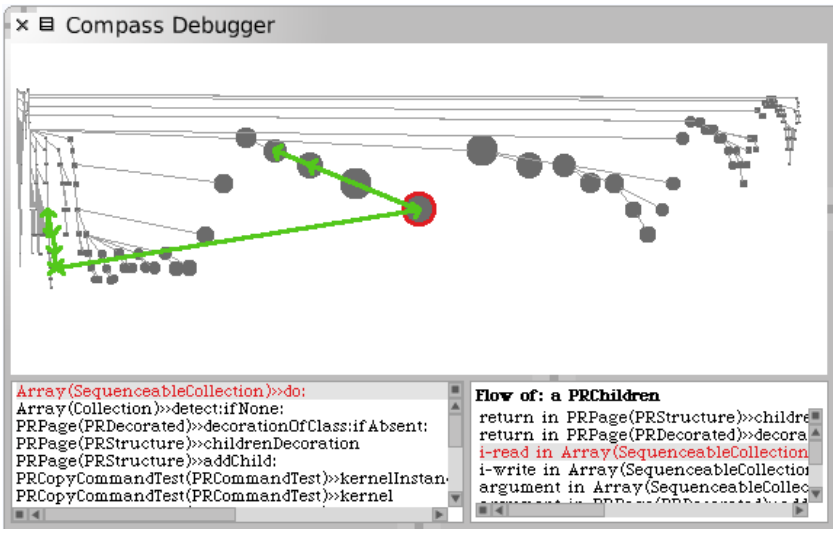

Fig. 8. Tracing object flow with the Compass back-in-time debugger

Changeboxes. Change management is an essential task to support software evolution. Smalltalk provides a simple form of change management already within the environment, so it is always possible to roll back changes. All changes are also logged, so it is impossible to lose code.

This form of change management, however, is strictly limited to source code. In a complex and evolving software system, different parts may depend on different versions of the same software base. For a system that cannot afford significant down-time, it may be unrealistic to expect that the entire code base be globally consistent at all times. *Changeboxes* [12] is a prototype of a system supporting change management for running software — deployed and development branches may co-exist in the same running image, and can run different versions of the same software. Branches can be dynamically split and merged without disrupting running clients, since the scope of applicable changes (*i.e.,* a "changebox") is always uniquely defined for any given context. A running web application, for example, can be modified without impacting clients, and incrementally deployed on the live system by merging branches when they are ready (Figure 9).

The Changeboxes prototype adapted the Smalltalk meta-model by modifying tools to be changebox-aware, and by modifying method lookup to select the right version of a method for the currently active changebox.

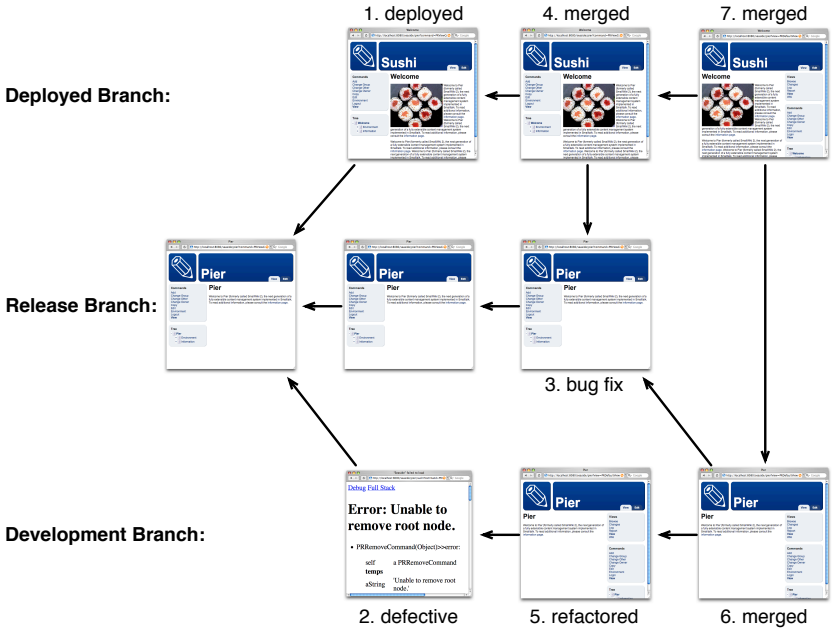

Fig. 9. Multiple versions of the same running system can be dynamically updated, split, and merged.

Reflectivity. Reflectivity [11] goes a step further in providing a general infrastructure for adapting running code at a fine level of granularity. Code is reified by its abstract syntax tree (AST), and links are installed on this representation as annotations (Figure 10). A compiler plug-in transforms the annotated ASTs before execution to take the links into account. When the annotated code is run, if any optional activation conditions are fulfilled, a message is sent to a designated meta-object to take appropriate action.

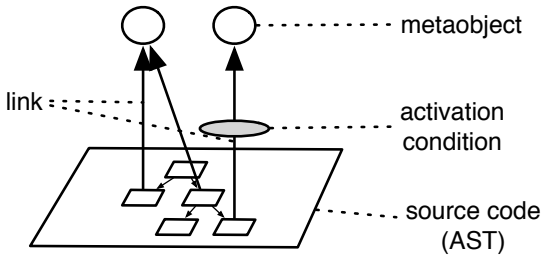

Fig. 10. Links annotate code reified as ASTs to trigger meta-object adaptations.

A typical application is to dynamically add and remove instrumentation code on a running system to gather statistics for program analysis [40]. Other applications, however, include aspect-oriented adaptation [42] and automatic adaptation of methods to use software transactional memory [39].

6 The Future of Change

Successful software must change to maintain its value. Why is it that the languages and environments we use to develop software inherently *inhibit* change rather than enable it?

We have seen how a simple object model which uniformly reifies all entities of the run-time meta-model supports dynamic change in a system like Smalltalk. Still, there are many aspects of software evolution that are no better handled by Smalltalk than by many other systems, both mainstream and exotic. Let us have a brief look at three of these issues.

Closing the gap between objects and models. Model-driven development (MDD) enables change by generating code from high-level models. When the models change, the corresponding code can be freshly generated. Models, however, are normally absent as artifacts in the running system, so no further changes are possible in a deployed system. Traditionally models, meta-models and meta-meta-models are distinct and their instances do not exist as entities at the same level. In Smalltalk-like systems, however, everything is an object, so objects, classes and metaclasses (for example) are all objects. They are all causally connected, so a change to a class impacts its instances, just as a change to a metaclass will impact the class.

Ultimately, *programming is modeling*, and a programming language or system is essentially a modeling tool. Object-oriented languages are particularly well-suited for allowing developers to design their own high-level models for a given application domain. An explicit notion of a model, however, is conspicuously missing from programming languages, Smalltalk included.

In the 1950s, FORTRAN was proposed as a high-level language from which computer code would be automatically generated. Nowadays we are used to thinking of programs written in high-level languages as being "the code", and we barely concern ourselves with the machine code that is "generated". By the same token, perhaps we should stop thinking about "generating code from models" and instead target development platforms where models themselves are executable. (Whether models are interpreted or code is generated on the fly should be purely an implementation detail.)

We have argued that change-enabled systems should be model-centric, making models available at run-time [35]. How to achieve this, however, is an open question, though some trends are interesting to watch. Executable UML [30] aims at making UML diagrams executable by means of dedicated compilers (though such models won't be available at run-time). Visual languages come and go [9], but some recent developments, such as Subtext [17], approach the

direct manipulation of models. Naked Objects [37] pushes ideas implicit in the model-view-controller paradigm to nearly eliminate the distinction between domain objects, their implementation and their view.

Eliminating the barrier between the image and the VM. Smalltalk objects live in the "image", a persistent representation of object memory. Images are saved as binary files, making it is easy to take multiple snapshots of the state of the system, move images between machines, and share images with other users. The virtual machine abstracts from the underlying hardware, so the same image can run on any hardware or operating system platform.

On the other hand, images are essentially single-user (even if they host web services), and communicating with the outside world (files, servers, other running images) is clumsy at best. Although advanced collaborative tools exist [41,7], objects by and large are "trapped in the image". Little work has been done recently to enable distributed, collaborative development.

Furthermore, although nearly everything is available to the run-time system, objects of the VM are not. There exists a hard barrier between the image and the VM which cannot be overcome. Certain kinds of changes are only possible by implementing a new VM. As we have seen in Section 5, object-flow analysis extended Smalltalks meta-model by introducing first-class aliases, but this was only possible by modifying the Smalltalk VM. Making such functionality available to other users is a non-trivial engineering task, since it is not simply a matter of loading a new package into the image.

To better support such deep changes in the run-time of change-enabled systems, we must either find ways to bridge the boundary between the image and the VM, or we need to erase the boundary completely. Pinocchio[7] [44] is an open language and system whose semantics and implementation is fully bootstrapped, allowing deep changes to be made at run-time. By eliminating the separation between the image and the VM, full control over the run-time semantics is possible. Non-intrusive changes important for software evolution, such as tracking object-flow or monitoring run-time performance can be dynamically enabled without requiring the VM to be replaced.

Putting objects into context. Most languages and systems, including Smalltalk, assume that the world is consistent. We are forced to assume that a name means one thing, that a single version of any piece of software is deployed at a time, that types and interfaces are consistent. The real world is rife with inconsistency, yet we cope with it very well. Why can't our software systems?

We cope with real world inconsistency because we easily keep track of different contexts. How we behave, how we react to events, and how we present ourselves depends on a constantly changing context. Furthermore we generally have little difficulty in managing multiple contexts being active at the same time. (We can deal with family, friends, co-workers and strangers present in the same room.)

[7] `http://scg.unibe.ch/research/pinocchio`

The Changebox prototype described in Section 5 managed multiple deployment contexts for software by adapting Smalltalk's method lookup to take a particular kind of context (a "changebox") into account, but this only works for changebox-aware tools. In practice, there are many different kinds of context variables that context-aware applications need to take into account [1,3]. To deal with context in a rigorous and fully general way, we argue that it is necessary to accommodate context deeply in the semantics [13] and the design [20] of programming languages and systems.

7 Conclusion

To sketch out what a change-enabled software system might look like, we have taken a brief look at a classic, dynamic system, Smalltalk, and seen how it supports software evolution in various ways. The single principle, *everything is an object*, can be seen as the driving force behind its simplicity, and its support for change. The key lessons for software evolution that we draw are: (i) *Less is more* — simple syntax and semantics can lead to a system that is easy to understand and change; (ii) *Reify everything* — by making all key entities first-class, they become available for modification and extension; (iii) *You can change a running system* — by causally connecting entities with their meta-descriptions, graceful, incremental change is enabled.

Change is here with us to stay. Increasingly, software systems will need to adapt to change dynamically, which means that software models must be accessible at run-time. Ideally, models will be executable, and different versions of the same models will need to be simultaneously active, and context-aware.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct 1, 2008 - Sept. 30, 2010) and the Hasler project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance" (project no. 2234). We also thank Lukas Renggli, Erwann Wernli, Fabrizio Perin, Bernhard Rumpe and Jan Ringert for their helpful comments and suggestions.

References

1. Gregory D. Abowd and Anind K. Dey. Towards a better understanding of context and context-awareness. In *Proceedings of the CHI 2000 Workshop on the What, Who, Where, When and How of Context-Awareness*. ACM Press, New York., 2000.
2. Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola — a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing — A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.
3. Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

4. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.
5. Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman, 2002.
6. G. Birtwistle, Ole Johan Dahl, B. Myhrtag, and Kristen Nygaard. *Simula Begin*. Auerbach Press, Philadelphia, 1973.
7. Avi Bryant. Monticello. http://www.wiresong.ca/Monticello.
8. Philipp Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, April 2009.
9. Margaret M. Burnett and Adele Goldberg. *Visual Object-Oriented Programming*. Prentice-Hall, 1995.
10. Damien Cassou, Stéphane Ducasse, and Roel Wuyts. Traits at work: the design of a new trait-based stream library. *Journal of Computer Languages, Systems and Structures*, 35(1):2–20, 2009.
11. Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.
12. Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007.
13. Mariangiola Dezani-Ciancaglini, Paola Giannini, and Oscar Nierstrasz. A calculus of evolving objects. *Scientific Annals of Computer Science*, XVIII:63–98, 2008.
14. Stéphane Ducasse, Tudor Gîrba, Adrian Kuhn, and Lukas Renggli. Meta-environment and executable meta-language using Smalltalk: an experience report. *Journal of Software and Systems Modeling (SOSYM)*, 8(1):5–19, February 2009.
15. Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.
16. Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, March 2006.
17. Jonathan Edwards. Subtext: uncovering the simplicity of programming. In Ralph Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2004, San Diego, CA, USA*, pages 505–518. ACM, 2005.
18. Martin Fowler. FluentInterface, on Martin Fowler's blog, December 2005. http://www.martinfowler.com/bliki/FluentInterface.html.
19. Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.
20. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.
21. C. A. R. Hoare. Hints on programming language design. Technical Report CS-TR-73-403, Stanford University, 1973.
22. Alan C. Kay. A personal computer for children of all ages. In *Proceedings of the ACM National Conference*. ACM Press, August 1972.
23. Alan C. Kay. The early history of Smalltalk. In *ACM SIGPLAN Notices*, volume 28, pages 69–95. ACM Press, March 1993.
24. Adrian Kuhn and Toon Verwaest. FAME, a polyglot library for metamodeling at runtime. In *Workshop on Models at Runtime*, pages 57–66, 2008.

25. Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.
26. Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.
27. Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz. Flow-centric, back-in-time debugging. In *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009*, volume 33 of *LNBIP*, pages 272–288. Springer-Verlag, 2009.
28. Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.
29. Kazutaka Maruyama and Minoru Terada. Debugging with reverse watchpoint. In *Proceedings of the Third International Conference on Quality Software (QSIC'03)*, page 116, Washington, DC, USA, 2003. IEEE Computer Society.
30. Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, May 2002.
31. Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.
32. Oscar Nierstrasz. Software evolution as the key to productivity. In A. Knapp M. Wirsing and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future*, volume 2941 of *LNCS*, pages 274–282. Springer-Verlag, 2004.
33. Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gaelli, and Roel Wuyts. On the revival of dynamic languages. In Thomas Gschwind and Uwe Aßmann, editors, *Proceedings of Software Composition 2005*, volume 3628, pages 1–13. LNCS 3628, 2005. Invited paper.
34. Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, Adrian Lienhard, and David Röthlisberger. Change-enabled software systems. In Martin Wirsing, Jean-Pierre Banâtre, and Matthias Hölzl, editors, *Challenges for Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, pages 64–79. Springer-Verlag, 2008.
35. Oscar Nierstrasz, Marcus Denker, and Lukas Renggli. Model-centric, context-aware software adaptation. In Betty H.C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 128–145. Springer-Verlag, 2009.
36. Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.
37. Richard Pawson. *Naked Objects*. Ph.D. thesis, Trinity College, Dublin, 2004.
38. Lukas Renggli, Stéphane Ducasse, and Adrian Kuhn. Magritte — a meta-driven approach to empower developers and end users. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 106–120. Springer, September 2007.
39. Lukas Renggli and Oscar Nierstrasz. Transactional memory in a dynamic language. *Journal of Computer Languages, Systems and Structures*, 35(1):21–30, April 2009.
40. David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Exploiting runtime information in the IDE. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008)*, pages 63–72, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

41. David A. Smith, Alan Kay, Andreas Raab, and David P. Reed. Croquet, a collaboration system architecture. In *Proceedings of the First Conference on Creating, Connecting and Collaborating through Computing*, pages 2–9, 2003.
42. Anselm Strauss. Dynamic aspects — an AOP implementation for Squeak. Master's thesis, University of Bern, November 2008.
43. Clemens A. Szyperski. *Component Software*. Addison Wesley, 1998.
44. Toon Verwaest and Lukas Renggli. Safe reflection through polymorphism. In *CASTA '09: Proceedings of the first international workshop on Context-aware software technology and applications*, pages 21–24, New York, NY, USA, 2009. ACM.