ESUG 2005 International Smalltalk Conference

Editors: Stéphane Ducasse, Serge Stinckwich

published as Technical Report
of the Institut für
Informatik und Angewandte Mathematik
University of Bern, Switzerland

iam-05-001

September 16, 2005

Abstract

Classification: 68-02 Research Exposition, 68N30 Mathematical aspects of software engineering (specification, verification, metrics, requirements, etc.) [New MSC2000 code] 68U35 Information systems (hypertext navigation, interfaces, decision support, etc.) [New MSC2000 code] D.2 Software Engineering, D.2.2 Tools and Techniques, D.2.7 Distribution and Maintenance. D.3.1 [Programming Languages]: D.3.2 Language Classifications, D.3.3 Language Constructs and Features (E.2), 68N15 Programming languages, 68N19 Other programming techniques (object-oriented, sequential, concurrent, automatic, etc.) [New MSC2000 code]

Table of Contents

These days, it seems that programming languages are really catching on both in the academic and industry worlds as popular dynamic programming languages. Dynamic programming languages are programming languages in which programs can easily modify their structure and their behavior as they run: new classes may be created, new modules may appear, software can be adapted very easily to new situation or needs, ... Most of the time, these languages are also dynamically typed, which some static typing advocates consider a drawback. However, according to advocates of dynamic programming languages, the flexibility of dynamic languages offsets these drawbacks, and even provides advantages so considerable as to make this an essential feature, for example for interactive programming or software evolution. Dynamic languages features (like reflection, dynamic reconfigurability, hot code replacement, software adaptation, ...) are almost essential to develop certain applications related to web, mobility, ambient computing, multi-agent approach and more generally soft computing. Even the latest mainstream static languages, have adopted to a certain extent dynamic language features such as garbage collection and limited forms of reflection.

Smalltalk is one of the first dynamic programming langages and still has unique features (fully object-oriented, minimal syntax, big open-source library of classes, reflective environment, ...) that superceed most of the current popular dynamic languages. Smalltalk is an object-oriented, dynamically typed, reflective, programming language designed at Xerox Palo Alto Research Center by Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg, and many others during the 1970s. The language was generally released as Smalltalk-80 and has been widely used since in many flavors.

Smalltalk is still the vehicle of choice for many software innovations and with the revival of dynamic languages, it should still regain interest among all those which wish to test ideas quickly within the context of the design of new programming languages, new IDEs or class framework, new way to design software (XP or test-driven for example). With Smalltalk, one got an alive and evolutionary environment that the developer and the researcher can quickly adapt to their needs.

You will found here a selection of papers presented at the Smalltalk Conference of the yearly Smalltalk event organized by ESUG - the European Smalltalk Users Group[1]. For the last three years, the goal of the Smalltalk Conference is to give wide academic recognition to high-quality research done in or with Smalltalk. The topics we are interested in are (in a non-exhaustive list): new languages features (mixins, AOP,...), meta and reflective programming, code analysis (refactoring,...), process development (Agile processes, Unit testing, ...), virtual machines (optimization, new trends, ...), frameworks (web, graphical...), software evolution (metrics, ...).

Each paper was reviewed by 5 members of the following international program committee:

- Pascal André (Université de Nantes, France)
- Noury Bouraqadi (École des Mines de Douai, France)
- Pierre Cointe (École des Mines de Nantes, France)

[1] http://www.esug.org/

- Wolfgang De Meuter (Vrije Universiteit Brussels, Belgium)
- Serge Demeyer (University of Antwerpen, Belgium)
- Stéphane Ducasse (University of Berne, Switzerland, Universit é de Savoie, France)
- Robert Hirschfeld (DoCoMo Euro-Labs, Germany)
- Alan Knight (Cincom Systems, USA)
- Thomas Kühne (Technische Universität Darmstadt, Germany)
- Michele Lanza (University of Lugano, Switzerland)
- Michele Marchesi (University of Cagliari, Sardinia, Italy)
- Kim Mens (Université de Louvain la Neuve, Belgium)
- Jean-Franǫis Perrot (Université de Paris 6, France)
- Bernard Pottier (Université de Bretagne Occidentale, France)
- Nathanael Schärli (University of Berne, Switzerland)
- David Shaffer (University of Wesminster, USA)
- Serge Stinckwich (Université de Caen, France)
- Roel Wuyts (Université Libre de Bruxelle, Belgium)

Papers accepted for the ESUG 2005 International Smalltalk Conference give a glimpse of high quality work conducted using Smalltalk. The spectrum of these research projects ranges from new programming concepts based on AOP or reflection to applications in various domains. We have selected the best five papers for a special issue in the journal : "Computer Languages, Systems and Structures". The current proceedings contains the eight papers accepted for the ESUG Research Track held at Brussels the 16th August 2005.

Session: Aspect-Oriented Programming

- "Open Aspects", Robert Hirschfeld and Stefan Hanenberg,
- "Towards Unified Aspect-Oriented Programming", Noury Bouraqadi, Abdelhak Seriai and Gabriel Leblanc,

Session: Reflection and Code Transformation

- "Inter-Langage Reflection – A Conceptual Model and Its Implementation", Kris Gybels, Roel Wuyts, Stéphane Ducasse, Maja D'Hondt,
- "Runtime Bytecode Transformation for Smalltalk", Marcus Denker, Stéphane Ducasse, Éric Tanter,

Session: Software Maintenance

- "Towards a Taxonomy of SUnit Tests", Markus Gälli, Michele Lanza and Oscar Nierstrasz,
- "Co-evolving Code and Design with Intensional Views – A Case Study", Kim Mens, Andy Kellens, Frédéric Pluquet and Roel Wuyts,

Session: Applications

- "A New Object-Oriented Model of the Gregorian Calendar", Hernán Wilkinson, Máximo Prieto and Luciano Romeo,
- "Microprints: A Pixel-based Semantically Rich Visualization of Methods", Romain Robbes, Stéphane Ducasse and Michele Lanza.

After reading those papers, we hope to see you at next ESUG International Smalltalk Conference.

September 2005
Stéphane Ducasse and Serge Stinckwich.

Open Aspects

Robert Hirschfeld[a,*], Stefan Hanenberg[b]

[a]DoCoMo Euro-Labs, Future Networking Lab, Landsberger Strasse 312, 80687 Munich, Germany

[a]University of Duisburg-Essen, Department of Computer Science, Schützenbahn 70, 45117 Essen, Germany

Abstract

Open Aspects are our approach to face unplanned changes in systems that are based on aspect-oriented composition at runtime. They support explicit adaptation models, allowing developers to describe system change events to be observed, and corrective actions to be taken. These events and actions cover both the base system affected by aspects as well as the aspects affecting the base system themselves. The proper combination of change events and corrective actions allows for conditional just-in-time runtime re-composition. This paper offers a detailed discussion of difficulties related to change in aspect-oriented systems and a description of consistency constraints inherent to them. An implementation illustrating Open Aspects and their application is provided.

Keywords: Aspect-oriented programming; dynamic aspects; open aspects; runtime weaving

1. Introduction

Systems utilizing aspect-oriented programming (AOP) differ in how and when they carry out the processes of composing aspects into the base system, a process also known as weaving. There are systems that statically weave at compile or load-time. Other systems permit the composition of aspects at an application's runtime.

We usually prefer to carry out changes to our systems while they are offline so that the detection and resolution of problems that become apparent during aspects composition will not interfere with the running system. However, there are situations where such practice is not desirable, is inadvisable, or even impossible due to domain specific requirements to the system in question. In telecommunications, for instance, system downtime results in disruption of services, leading to less customer satisfaction, and because of that has to be kept to a minimum or to be avoided entirely [13,14,16]. Ambient and embedded computing infrastructures and environments are yet another example of systems that require online adaptations – changes to the running system. The reason for weaving in aspects dynamically results from the requirement that the system aspects to be woven are expected to change at runtime.

Recent work of the aspect-oriented software development community indicates that dynamic aspects are becoming of increased interest. Dynamic aspects offer compositions that can be made effective or revoked at runtime. Prominent activities in this research area are efforts to provide technologies for dynamic method call interception (MCI, [19]) or extensions to virtual machines (VM) for enhanced method call dispatch. Systems like PROSE [23,24], Steamloom [3], JAC [22], AspectL [8], or AspectS [12] are all concerned with hot-deployment of aspects. They employ runtime weaving to dynamically add new code, modify or remove available code or change the way the base application is interpreted. Their weaver considers the systems or code segments to be combined at one particular point in time. Pointcut expressions or predicates (both terms are used interchangeably in this text) are evaluated to compute sets of join-point shadows [20] to be instrumented, and integration steps are performed as necessary to provide the desired composed behavior. Join-point shadows are roughly correspondent

* Corresponding author. Tel.: +49-160-4785212; fax: +49-89-56824-300; e-mail: hirschfeld@acm.org.

locations of actual join-points in a program's representation such as the program's source or its meta structure. We consider poincut predicates to be one essential contribution of AOP to generically designate subsets of a system's computational properties. Examples for that are all accesses to an instance variable, all sends are receptions of a message, or all messages sent by a group of senders or received by a specific receiver.

Opening up systems and allowing them to be changed after their initial deployment – possibly by code providers other than the original one and probably while they are running – increases the likelihood of system changes not planned for at the beginning of their design and development. Furthermore, in such open systems the point in time changes can happen as well as their order and frequency are undetermined. Presence and characteristics of classes, instances, or methods can be revised at any instant or not at all. Furthermore, we can assume changes that not only address elements that belong to the base system, but to affect aspects themselves. Changing pointcuts and their associated sets of join-point shadows or changes to advice code is an example.

Changes like that can and will have an effect on AOP-induced invariants as mentioned above. Hence, a mechanism is needed to explicitly maintain these invariants.

As a simple illustration let us look at a system that uses classes not known at compile time but loads them dynamically on demand. New classes not known during the development and initial deployment of the original system thus appear. In such a situation the problem from the aspect-oriented point of view is that it is not clear how the system should behave. Should pointcut coverage be monitored, and, if necessary, should aspects be recomposed? Or should such changes be ignored at all? Questions like that are not addressed by current approaches and technologies.

Open Aspects is our approach to handling unplanned system changes at runtime. Open Aspects support explicit adaptation models, allowing developers to describe system change events to be observed and corrective actions to be taken in response to these events. System change events and corrective actions cover both the base system affected by aspects and the aspects themselves affecting the base system. The proper combination of change events and corrective actions allows for conditional just-in-time runtime re-composition.

Contributions of our paper include:

- A description of consistency constraints inherent to aspect-oriented systems
- A detailed discussion of the associated change problem
- A solution to this change problem by separating and providing an explicit adaptation model to aspect developers
- An implementation illustrating our solution and its application

In the next section we give a motivating example. In Section 3 we explain Open Aspects in general. Section 4 demonstrates how Open Aspects work in the presence of change. Section 5 describes OpenAspectS, our implementation of Open Aspects in AspectS. Section 6 shows an application example of OpenAspectS. After discussing related work in section 7, we summarize our paper in section 8 and come to a conclusion.

2. Motivation

2.1. Aspect Composition Models

In most aspect-oriented systems there is typically a weaving mechanism that composes aspects and the base system they apply to according to descriptions offered by pointcut predicates or expressions. Usually, such a composition is initiated by developers at a particular point in time. This point in time might vary from development-time, over compile- and load-time, up to runtime. Here it is important to note that each and every composition is either carried out implicitly by development tools or explicitly by instructions stated explicitly in the flow of control of the running system. This process can be characterized as one-time model composition.

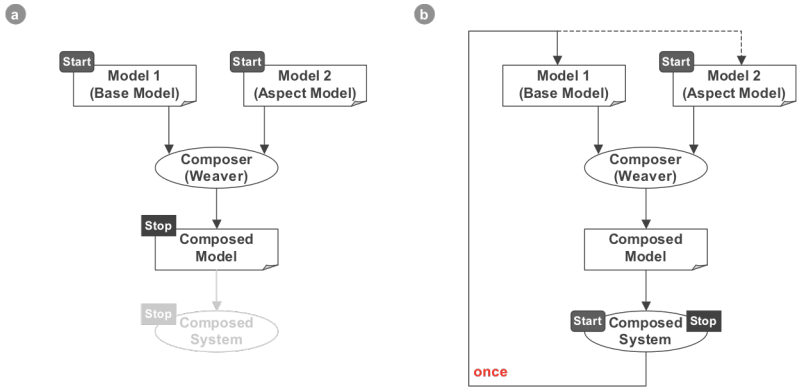

Fig. 1. Static (a) and dynamic (b) one-time composition

Fig. 1-a illustrates static one-time model composition. Weaving starts from both a model of the base system and the aspect model (as indicated by the 'start' tags). A composer then combines both these models into a composed model. For that, all join-point shadows involved are instrumented so that the composed model shows all desired properties (marked with a 'stop' tag). Next the composed model gets effective in the composed system at runtime. Note that in all figures showing composition models (Fig. 2,1,11) the dog-eared rectangles represent passive models, whereas the ovals represent active ones, that is, system parts being executed.

In Fig. 1-b we can see an extension of the previously described process for dynamic one-time model composition. Now we start off from a running system and an aspect model. The weaver derives the base system's model from the running base system and then composes this derived and the aspect model into the composed model which in turn will be made effective in a new version of the running composed system. Since developers can initiate weaving at any point in runtime, we can treat the initial base system as a special case of the composed system.

Even though this procedure can be performed repeatedly, we still characterize it as one-time since the injection or removal of join-point shadows is set off by an explicit activity in the development process or program expression at load or runtime. In both cases of one-time weaving, the weaver only considers join-point shadows described by all involved pointcut expressions or predicates at that particular point in time. Future changes to the system that were not planned for, both to the base system and to the set of incorporated aspects, cannot be considered properly or at all.

Even with continuous weaving, presented in our work on Morphing Aspects [11], there are changes not taken into account. As shown in Fig. 2, morphing aspects constantly derive a minimal base model needed to inject or remove join-point shadows necessary in the immediate future.

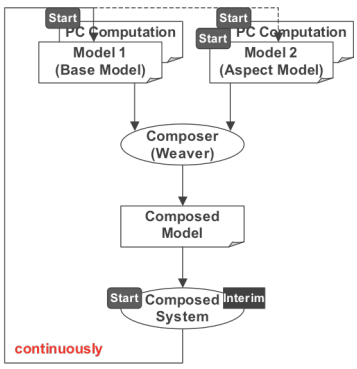

Fig. 2. Dynamic continuous composition

While the weaver for every new step examines the current system, all previous system compositions are not revisited for changes that might have had an influence on those earlier compositions.

In the following we provide an example illustrating these effects.

2.2. Running Example

Consider the following example as described in [12]. It is written in AspectS [12], a general-purpose dynamic AOP environment for Squeak/Smalltalk [15][1]. Squeak is an open and highly portable implementation based on the original Smalltalk-80 system [10]. Here we want to monitor all mouseEnter: and mouseLeave: messages received by instances of Morph (a base class in Morphic, a user interface framework of Squeak) and its subclasses by logging them to the system transcript, Smalltalk's equivalent to a system console (Fig. 3).

```
MorphicMousingAspect>>adviceMouseEnter

  ↑ BeforeAfterAdvice
    qualifier: (AdviceQualifier
      attributes: { #receiverClassSpecific. })
    pointcut: [
        Morph withAllSubclasses
      select: [:m | m includesSelector: #mouseEnter:]
      thenCollect: [:m | AsJoinPointDescriptor
        targetClass: m targetSelector: #mouseEnter:]]
    beforeBlock: [:receiver :arguments :aspect :client |
    Transcript show: '*Enter*', arguments first printString]

---------------------------------------------------------

"aspect lifecycle in a nutshell"
| anAspect |
anAspect ← MorphicMousingAspect new.
anAspect install.
anAspect uninstall.
```

Fig. 3. Advice and lifecycle example

We employ an aspect called MorphicMousingAspect to trace the reception of these messages. Advice code to trace the reception of mouseEnter: and mouseLeave: messages is stated in two advice methods adviceMouseEnter and adviceMouseLeave. Each advice method creates a BeforeAfterAdvice object that allows us to state behavior before and after the invocation of a method. Once the advice object is created, it is further qualified via the #receiverClassSpecific advice qualifier attribute causing the advice code to be executed for all message receivers described by the pointcut expression. In our example from Fig. 3, these are all instances of Morph and its subclasses responding to mouseEnter:. In adviceMouseEnter join-point descriptors are collected by querying the system for all classes that are subclasses of Morph and implement mouseEnter:. The block to be executed before the actual invocation of mouseEnter: echoes the event passed with the mouseEnter: message to the transcript. An adviceMouseLeave advice works likewise for the reception of mouseLeave: messages. To activate the MorphicMousingAspect, one creates an aspect instance and sends it an install message when desired. An installation activity can be considered atomic. In a plain Squeak image (version 3.6) there are 24 implementers of a method named mouseEnter: and 21 implementers of a method named mouseLeave:. 22 of the 24 of mouseEnter: methods and 19 of the 21 of mouseLeave: methods are found in Morph and its subclasses. Our instance of MorphicMousingAspect, when installed, instruments all 22 + 19 = 41 locations.

Fig. 4 shows a subset of the Morph class hierarchy emphasizing some of the classes affected by the installation of MorphicMousingAspect by rendering them with a squared texture. All classes with behavior not influenced by the installation of the aforementioned aspect are displayed as circles with a white filling area. Little circular symbols on top of circles representing classes mark them as holding join-point shadows that belong to the pointcut of the aspect under consideration. Here, Morph, MenuItemMorph, and WonderlandCameraMorph are marked as being involved in MorphicMousingAspect.

[1] Readers not familiar with Squeak/Smalltalk but articulate in Java might find the language syntax comparison in [7] of help when examining code fragments throughout the paper.

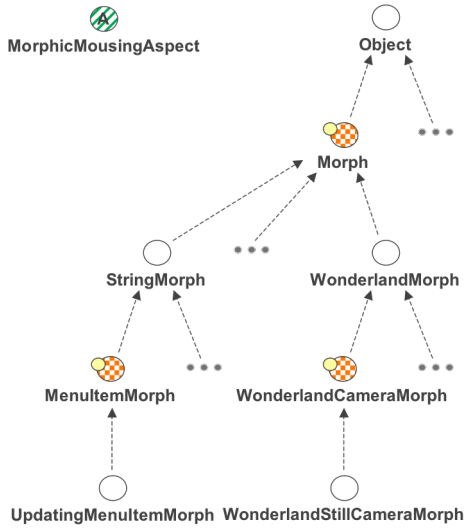

Fig. 4. Visualization of class hierarchy with installed aspect

2.3. Changing the Base System

What happens if, after the installation of MorphicMousingAspect as described above, a new class MouseEnterLeaveMorph is added to our system (Fig. 5)? MouseEnterLeaveMorph is a subclass of Morph and also re-implements mouseEnter: and mouseLeave:. By doing so, this class would have been part of the set of join-point descriptors computed by the evaluation of our pointcut expressions and instrumented by our weaver. However, when our aspect was installed this class was not yet present in our system and was thus not considered during weaving.

The result of adding this class after installing our aspect is shown in Fig. 6. Even though the evaluation of the pointcut expression would include the newly added class and its mouseEnter: and mouseLeave: methods (as indicated by the little circular symbol added to the circle representing MouseEnterLeaveMorph), the class as such remains unaffected by our aspect (as suggested by the white fill color of the class symbol). Our installed aspect only covers the 41 locations computed during weaving, leaving the new additional two locations unaffected.

```
Morph subclass: #MouseEnterLeaveMorph
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries:  ''
   category:  'AspectS-Examples'

MouseEnterLeaveMorph>>handlesMouseOver: evt
   ↑    true

MouseEnterLeaveMorph>>mouseEnter: evt
    self   beepPrimitive.

MouseEnterLeaveMorph>>mouseLeave: evt
    self   beep.
```

Fig. 5. Code of newly added Morph subclass

Depending on specific application scenarios, these two missing join-points shadows might or might not cause erratic system behavior. Leaving them unaffected would honor the intent to only weave-in all join-point shadows covered by the aspect's join-point expression at installation time. On the other hand, adjusting them automatically would ensure the consistent application of the aspect to all join-point shadows covered by its join-point expression.

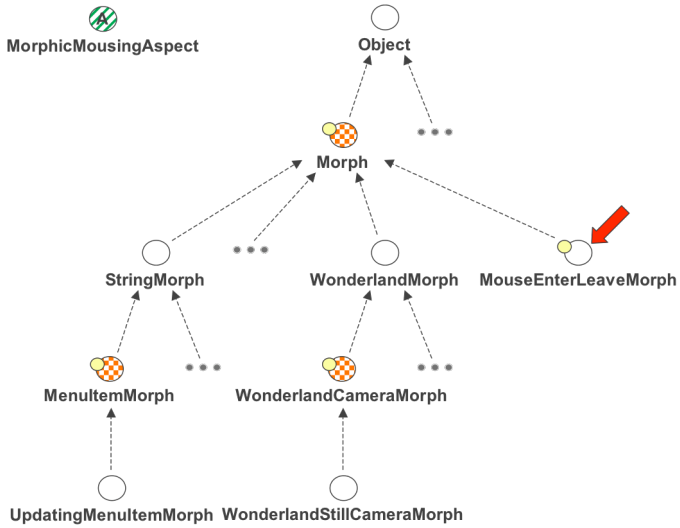

Fig. 6. Visualization of aspect composition after the addition of the new subclass of Morph

The described change scenario is an additive one. Subtractive changes to the base system have similar effects.

2.4. Changing an Advice's Pointcut

What happens if, after the installation of MorphicMousingAspect as described above, the pointcut expression of the installed aspect is changed? What if the pointcut expression of MorphicMousingAspect is, for example, modified to collect all join-point descriptors by querying the system for all implementers of mouseEnter: and mouseLeave: at StringMorph which is a subclass of Morph instead at Morph itself (Fig. 7)?

```
...
pointcut: [
    StringMorph withAllSubclasses
        select: [: m | m includesSelector: #mouseEnter:]
        thenCollect: [: m | AsJoinPointDescriptor
            targetClass: m targetSelector: #mouseEnter:]]
...
```

Fig. 7. Changed pointcut expression of installed aspect

Changing our pointcut expression in such a way reduces the number of join-point descriptors contained in the set it evaluates to from 22 to one for mouseEnter:, and from 19 to one for mouseLeave: (Fig. 8). This leads to the same situation as described above where our pointcut became out of sync with the system actually instrumented in the process of installing the new class MouseEnterLeaveMorph.

Depending on specific application scenarios, the 39 join-point shadows that are still in the system but no longer covered by the aspect's pointcut expression might or might not cause erratic system behavior. Leaving them in the system would honor the intent to only weave-in join-point shadows covered by the aspect's join-point expression at installation time. On the other hand, removing them automatically would ensure the consistent application of the aspect to all join-point shadows covered by its join-point expression.

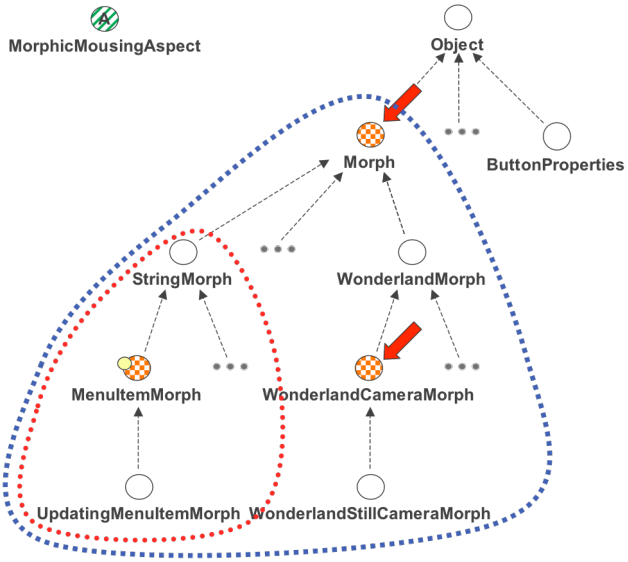

Fig. 8. Visualization of composition after changed pointcut coverage

2.5. Changing an Aspect's Advice

Also, consider what happens if, after the installation of MorphicMousingAspect, the advice code gets changed (Fig. 9)? The weaver previously used the advice code present during weaving to instrument the system. In our example, the weaver provisions advice code that logs the execution of mouseEnter: and mouseLeave: methods to the system transcript.

```
...
beforeBlock: [:receiver :arguments :aspect :client |
    Logger count increment.
    Transcript show: '*Enter*', arguments first printString]
...
```

Fig. 9. Changed advice code of installed aspect

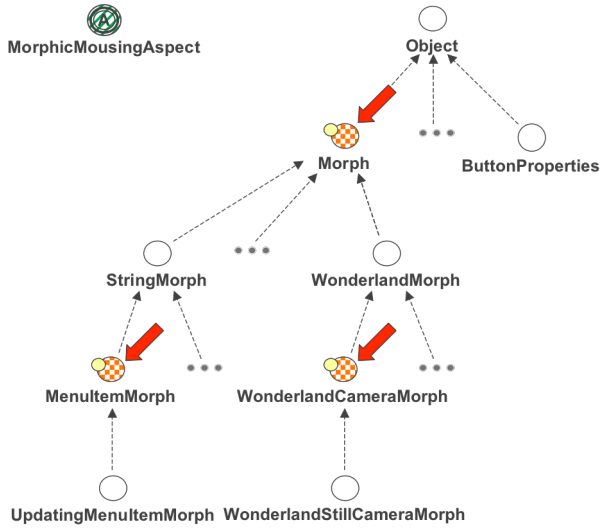

Fig. 10. Visualization of composition after changes to advice code

Changing that particular code located within our advice block leaves all 41 locations in our image that are instrumented with the previously available advice code out of sync with its current version. In Fig. 10 the different versions of the advice code residing in the aspect and composed into our system are expressed by different line styles of the circles representing our aspect and system classes.

Again, correct system behavior depends on particular application scenarios that might require the instrumented system parts to be updated or left as is. Instead of leaving the actual adaptation behavior to a particular weaving mechanism, we prefer to give the developer a means to explicitly decide about adaptations if demanded.

3. Open Aspects

3.1. System Changes

In open systems that are allowed to change at runtime, aspect composition needs to explicitly address changes to both the base system and the set of aspects that have been applied to it. These changes comprise added, transformed, or removed classes and methods. The addition, modification/transformation, or removal of pieces of advice associated with an aspect or pointcut expressions or predicates associated with a piece of advice also need to be handled.

A weaver determines all join-point shadows of all involved aspects by evaluating all associated join-point expressions or predicates. After completion, the set of join-point shadows available in the composed system correspond to the ones computed during weaving. Every change to the system has the potential to bring this correspondence out of sync. If not dealt with correctly, such inconsistency may or may not lead to erratic system behavior.

3.2. Change Events and Corrective Actions for Open Aspects

The abovementioned changes leading to composition inconsistencies can be addressed in quite different ways. Whether or not corrective actions taken to react on system change events will lead to expected system behavior can only be decided in a particular application context. While some applications do not expect changes to be considered after the composition of aspects, others might be required to adjust aspect composition accordingly. Here, we will describe corrective actions we consider important and feasible for operating open systems in use:

- None/indifferent
- Full reinstall
- Partial reinstall
- Partial withdrawal
- Full withdrawal
- Reject

'None/indifferent', not reacting to any change happening after the installation of an aspect, is the simplest corrective action that may be taken. 'Full reinstall' will cause the affected aspect with all its pieces of advice to be uninstalled and then and reinstalled again afterwards, fully applied to the newly inserted set of join-point shadows. 'Partial reinstall' will reinstall only pieces of advice affected by a change, leaving all unaffected advice installed and untouched. 'Partial withdrawal' will uninstall only pieces of advice affected by a change, leaving all unaffected advices installed and untouched. 'Full withdrawal' will cause all affected aspects with all their pieces of advice to be withdrawn from the system for good. 'Reject' will prevent attempts to change the base system affected by respective advice directives to get effective. While these corrective actions seem the most obvious to us, there are certainly other that could be added to this list.

3.3. Adaptation Models of Open Aspects

Adaptation models of Open Aspects are a means to explicitly provide corrective actions to be carried out in response to system change events. With adaptation models, Open Aspects allow for the flexible association of change events and corrective actions, according to specific needs of the application scenario to be supported and the system behavior to be achieved.

With Open Aspects there is an explicit separation of base, aspect, and adaptation models. This allows an explicit association of elements belonging to them. Such association expresses which elements of the base system need to be affected by which aspects and their associated pieces of advice under the occurrence of which set of change events. This lets us directly say how or if at all to react to system events. While this approach can certainly be extended to deal with any type of event, we will for now limit ourselves to system change events as described above.

3.4. Conditional Weaving with Open Aspects

We characterize the weaving model of our approach to Open Aspects as dynamic conditional model composition. As illustrated in Fig. 11 there is now, besides the base and the aspect model, also an adaptation model to be considered by the weaver when composing such models. As already described for dynamic one-time composition (in Fig. 1-b) and dynamic continuous composition (Fig. 2), we start from a composed system at runtime. The weaver initially derives a model of the running base system needed for making our aspect model effective (both marked with a 'start' tag). While doing so, the weaver also examines an adaptation model (also marked with a 'start' tag) detailing all involved system change events to be observed and all corrective actions to be taken in correspondence to the system elements involved as described above.

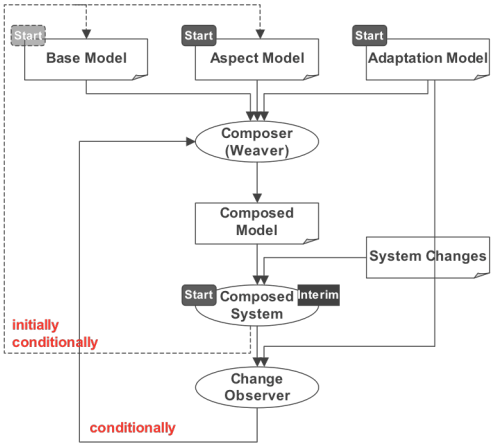

Fig. 11. Dynamic conditional composition

The weaver affects the composed system as in the other dynamic models described before. In addition to that it provides for change event handling. Such event handling can be supported in a variety of ways. One way to address system change events is the inlining of event handling code into the composed system that itself initiates specified corrective actions (amongst them system re-composition) in the event of change. Another way is the provisioning of a separate system entity we call a change observer that reacts to system change events appropriately. Compared with the previous method, the composed system does not include any code related to conditional model composition. Event propagation as such may be implemented in quite a few ways. Implementations can vary from a very simple and naïve propagation from an event source to an event consumer to quite sophisticated event filters preprocessing events according to complex rule sets.

4. Illustration

4.1. Starting Situation

In the following we will illustrate how Open Aspects behave in open systems. For that we will use the notation as shown in the right side of figure Fig. 12. In part (a), the left side of Fig. 12, we start out with two modules running on our platform with no aspects yet applied. This set of modules can be extended by adding new modules or curtailed by removing existing ones.

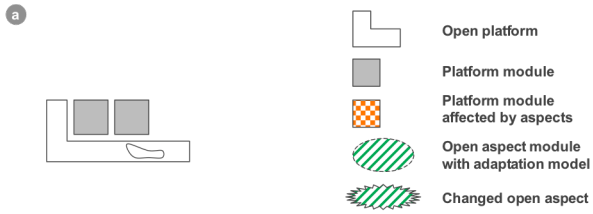

Fig. 12. Computational platform (with legend)

The L-shaped open platform represents that part of the runtime environment that remains quite stable over time with respect to change. Boxes represent modules that are expected to change. If affected by an open aspect, such a box is displayed with squared pattern texture, or is gray otherwise. Open Aspects are rendered as ovals. If an open aspect was changed, its boundary appears as a zigzag line.

4.2. Additive Changes to the Base

In Fig. 13 we demonstrate composition adaptation with respect to additive changes to the base system. In part (b) an open aspect gets applied to the system depicted in part (a) of Fig. 12.

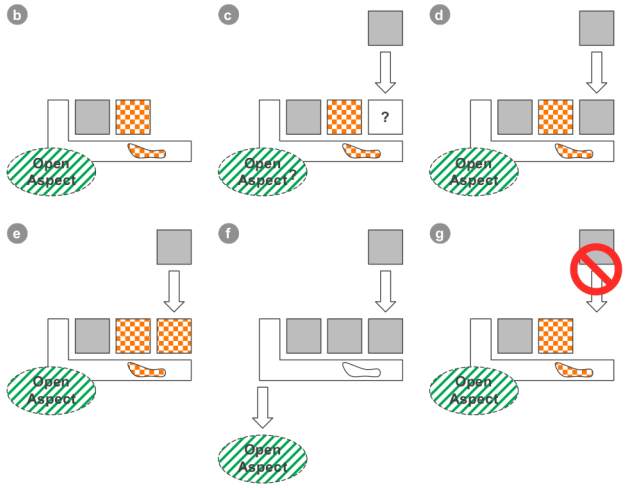

Fig. 13. Additive changes to the base

This application affects one module (the second box from the left) as well as part of the platform. Note that the effect of an open aspect to the platform itself is only shown to emphasize that Open Aspects can be applied to the platform itself as well and are not limited the user-supplied modules. Part (c) presents an additive change to be dealt with by our Open Aspects infrastructure. Here a third component is put on our platform, with an open aspect already installed. Parts (d) to (g) illustrate how the system might respond to such change according to the adaptation model associated with the open aspect applied.

In (d), we can see the most simple of all adaptation strategies in action: ‘none/indifferent’ which leaves the system as is, without taking any corrective action. Note that in this case the same behavior can be observed if there is an adaptation strategy applied other than ‘none/indifferent’, but that under the given circumstance the adaptation model does not require any corrective action to be taken. In (e), the newly added component requires some aspect-related composition and is adjusted accordingly. An adaptation model requiring a ‘partial or full reinstall’ could have instigated this behavior. The effect of executing a ‘full withdrawal’ can be seen in part (f). Part (g) shows that the Open Aspects infrastructure might as well refuse the addition of further components if required by an adaptation model.

4.3. Subtractive Changes to the Base

In Fig. 14 we explain composition adaptation in response to subtractive changes to the base system.

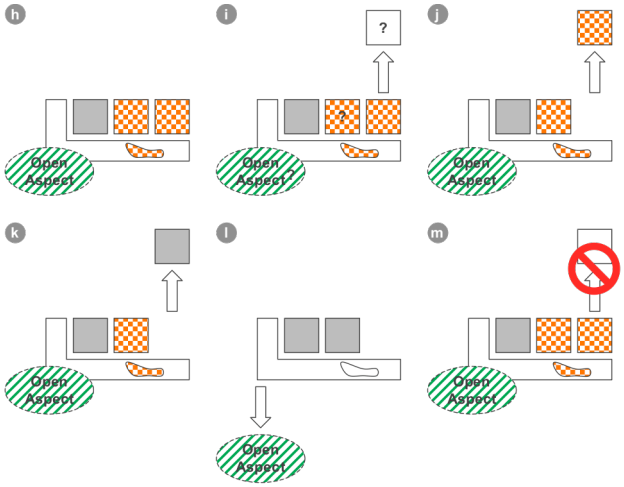

Fig. 14. Subtractive changes to the base

In (h) we start from a situation in which an open aspect applied to our system affects two of its three modules. We are now going to remove one module, as show in part (i). Parts (j) to (m) illustrate how the system might respond to such subtractive change according to the adaptation model associated with the open aspect applied.

In (j), the system is left as is, without taking any corrective action as a result of a 'none/indifferent' adaptation strategy. In (k) the removed component will be freed from all compositions by the applied Open Aspects that were effective previously. This, for example, can be the result of a 'partial withdrawal' or a 'full reinstall'. Reverting a removed module to the form it was in previously to its addition to the system can be of interest to modules that have to be moved to storage or into another execution context such as another system to enable a defined launch thereafter. The effect of executing a 'full withdrawal' as response to the subtractive change can be seen in part (l). Part (m) shows that the Open Aspects infrastructure might as well refuse the further removal of components if required by an adaptation model.

4.4. Aspect Changes (Pointcut and Advice)

In Fig. 15 we show how changes to an open aspect itself might affect all compositions originated by this aspect so far, according to the adaptation model associated with the open aspect installed. The aspect installed in (n) affects two of the three modules available on our platform. What happens to the system if this aspect (some or all of the advice code associated with it, or some or all pointcuts associated with its pieces of advice) is changed as indicated in (o)? Parts (p) to (s) describe corrective actions that could be taken to address such change to an open aspect.

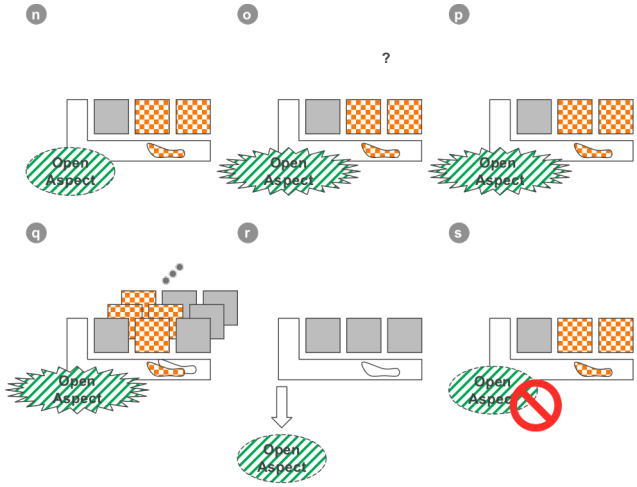

Fig. 15. Changes to the aspect itself

In (p), no corrective action is taken at all, leaving our system in the composition state as observed before the change to the open aspect. In (q), several possible combinations of changes are implied to the modules or the platform as a result of a 'partial withdrawal', or 'partial or full reinstall' if required by the adaptation model associated with the changed aspect. Similar to additive or subtractive changes to the base system, changes to an open aspect might result to either a 'full withdrawal' (r) or the refusal of the change itself (s).

5. Implementation

5.1. AspectS

AspectS extends Squeak to allow for experimental aspect-oriented system development. Its goal is to provide a platform for the exploration of dynamic late-bound aspect-oriented software composition. It employs coordinated meta-level programming to address the tangled code phenomenon. AspectS shows great flexibility by not relying on code transformations, but by making use of metaobject composition instead. AspectS provides a framework for developers to construct the proper runtime structure of aspect instances. Once instantiated, an aspect instance refers to its associated advice objects that maintain all information about what additional code (Computation, an instance of BlockContext) has to be performed where (Pointcut, an instance of BlockContext, to compute all shadow join-points to instrument) and when (described via AdviceQualifier attributes).

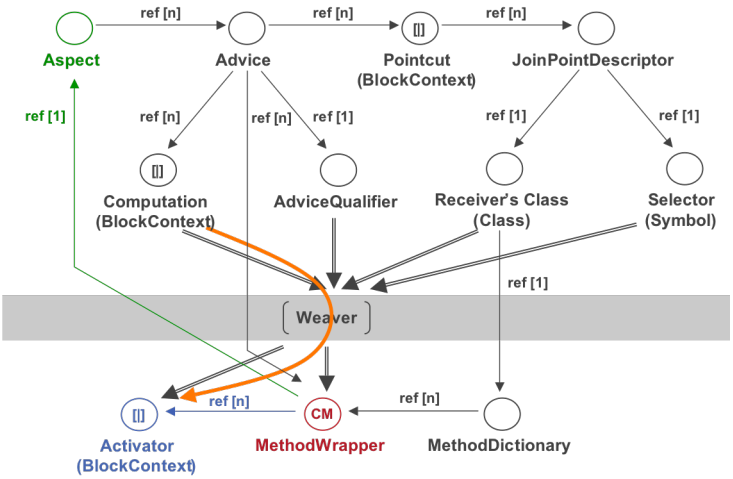

Fig. 16. Weaving in AspectS

Weaving or unweaving happens every time an install or uninstall message is sent to a respective aspect instance. Installation causes the pointcut computation to be executed returning a set of join-point descriptors indicating locations in the system structure to be affected. Then for each join-point descriptor the weaver creates an appropriate method wrapper [4] instance matching the advice type. Each such wrapper is then configured with the actual advice code as well as one or many so-called activation blocks. Activation blocks are selected according to advice qualifier attributes provided by the developers. They perform residual runtime tests deciding if the advice code a join-point shadow was instrumented with is going to be executed or not. Fig. 16 shows both the metaobject structure created by programmers using the AspectS framework, as well as the metaobject structure constructed or affected by the weaver when installing or uninstalling an aspect. Note that the latter object structure is based on the former.

5.2. OpenAspectS Extensions

OpenAspectS is our prototypical implementation of Open Aspects. It is an extension to AspectS. OpenAspectS is based on Squeak version 3.6 and AspectS version 0.5.4. OpenAspectS extends AspectS' basic runtime structure as shown in Fig. 17. We added an active pointcut (ActivePointcut) system element associated with each advice. An active pointcut object records the set of all join-point descriptors that were associated with that aspect when the installed aspect was woven into the system.

This set of join-point shadows is obtained by executing the pointcut expression (Pointcut) associated the respective advice, as described above.

In OpenAspectS, we added adaptation strategies (AdaptationStrategy) to advice qualifiers (AdviceQualifier) implementing corrective actions to be taken in the presence of change relevant to a particular advice or its associated aspect. In our current implementation we provide the following adaptation strategy examples:

- None/indifferent (via the #indifferent advice qualifier adaptation attribute)
- Full reinstall (via the #reinstallAspect advice qualifier adaptation attribute)
- Partial reinstall (via the #reinstallAdvice advice qualifier adaptation attribute)
- Partial withdrawal (via the #withdrawAdvice advice qualifier adaptation attribute)
- Full withdrawal (via the #withdrawAspect advice qualifier adaptation attribute)

The extended advice qualifier allows us to explicitly associate adaptation strategies with advice and aspects.

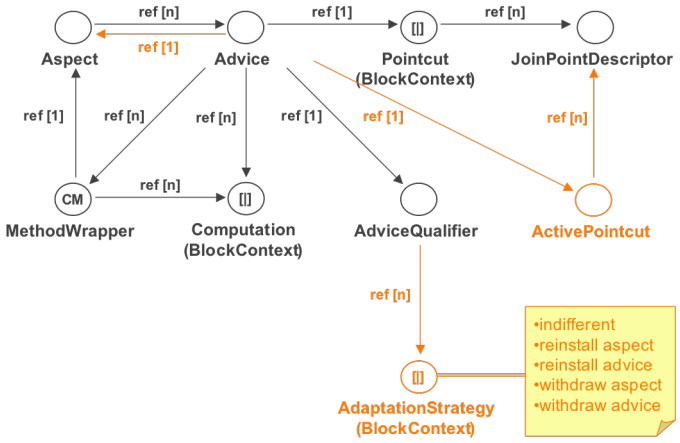

Fig. 17. OpenAspectS runtime structure extensions

According to the specified adaptation attributes, the weaver selects the appropriate adaptation strategies and initiates the registration of the aspect or advice instance with the change notification infrastructure.

5.3. Changes and Relevance Checks

In order to make Open Aspects aware of system change events of interest, we extended Squeak's base to provide us with change notifications for each change to a class or its methods. We modified Squeak to provide proper system change events, as well as with a single point of registration for such notifications. System change notifications are now supplied for each addition, transformation, or removal of individual classes or methods, similar to the dependent maintenance protocol of the CLOS Metaobject Protocol (MOP, [17]). Furthermore we provide such notifications right before and right after one of the aforementioned changes are carried out by the system. This gives us more flexibility and more accuracy in selecting suitable corrective actions.

We implemented a mechanism to determine if a change to the system indicated by a system change event is relevant to an individual advice or aspect (Fig. 18).

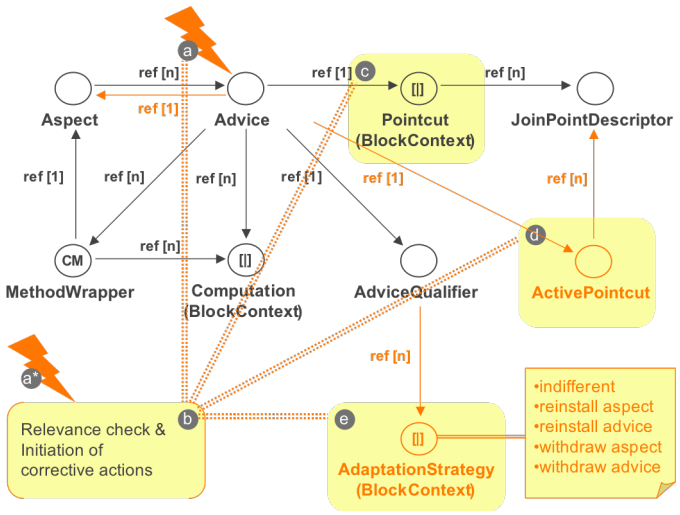

Fig. 18. OpenAspectS change events and relevance checks

When an aspect and all of its advice is installed, each advice registers for system change events listed above. If there is a change event, an advice that expressed interest is informed (a). In response to that, the determination of relevance is started (b). For this relevance check the pointcut expression is reevaluated (c). The newly computed result representing the set of join-point shadows that would be provisioned if the aspects would be installed now is then compared with the actual set of join-point shadows (stored as active pointcut) the aspect was associated during its actual installation (d). If the comparison indicates that the two sets are different and the adaptation model requires a corrective action to be taken, a corresponding adaptation strategy is selected and executed (e). Note that this is a simplified implementation to illustrate the basic flow of events. More complex systems might demand more complex and sophisticated event subscription and distribution mechanisms.

5.4. More Implementation Options

In another implementation we allowed the developer to explicitly associate change event types and corrective actions by accepting pairs of change and action descriptors as adaptation attributes in advice qualifiers. Examples of such pairs are #(changedAdvice fullReinstall), #(changedPointcut partialReinstall), and #(changedMethod indifferent). In our current implementation we decided, for pragmatic reasons, to treat all change events evenly and because of that we allow only action descriptors to be given as adaptation attributes in advice qualifiers.

6. Application

6.1. Starting Situation

Our Open Aspects example starts out similarly to the one described in section 2. This time we are utilizing the Open Aspects platform as shown in our code example. The main difference for the programmer is the additional adaptations attribute section for advice qualifiers.

In Fig. 19 we can see that the developer decided to let the advice being reinstalled in the event of a change to the base system that extends to the span of this piece of advice (adviceMouseEnter).

```
MorphicMousingOpenAspect>>adviceMouseEnter

  ↑ BeforeAfterAdvice
    qualifier: (AdviceQualifier
      attributes: { #receiverClassSpecific. }
    adaptations: { #reinstallAdvice. })
    pointcut: [
        Morph withAllSubclasses
      select: [:m | m includesSelector: #mouseEnter:]
      thenCollect: [:m | AsJoinPointDescriptor
        targetClass: m targetSelector: #mouseEnter:]]
    beforeBlock: [:receiver :arguments :aspect :client |
    Transcript show: '*Enter*', arguments first printString]
```

Fig. 19. Example advice in OpenAspectS

Since the pointcut expressions of adviceMouseEnter (as seen in Figs. 3 and 19) and adviceMouseLeave are the same as of MorphicMousingAspect, the activation of an instance of MorphicMousingOpenAspect instruments the same 41 locations in the image as well (Fig. 20).

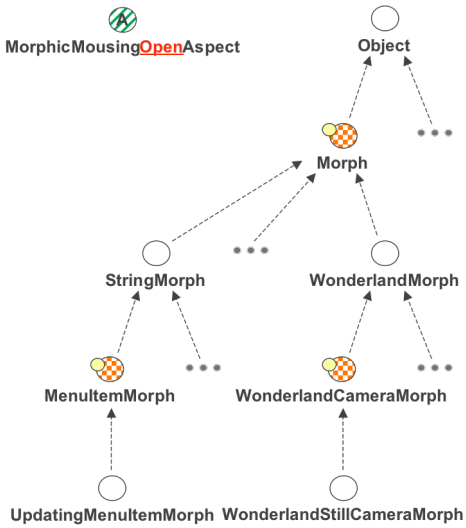

Fig. 20. Morph sub-hierarchy with installed open aspect

6.2. Changing the Base System

With an open aspect instance of MorphicMousingOpenAspect installed in our system, adding the new class MouseEnterLeaveMorph implementing mouseEnter: and mouseLeave: will change our system as shown in Fig. 21. Here the part of the class hierarchy framed with a box labeled *previously* denotes the observable effect prior to Open Aspects.

Our reinstall-advice adaptation strategy, selected in our advice qualifier by providing the #reinstallAdvice adaptation attribute, caused the Open Aspects environment to notice the addition of a new class, its having an effect to the pointcuts of adviceMouseEnter and adviceMouseLeave of MorphicMousingOpenAspect. As a response to this change, the two pointcut expressions are reevaluated and the pieces of advice associated with them are reinstalled.

Besides reacting properly on additive changes, transformative and subtractive changes need to be addressed appropriately as well. In the following we show how the removal of methods and classes covered by the pointcuts of our installed aspect instance of MorphicMousingOpenAspect can affect the aspect composition.

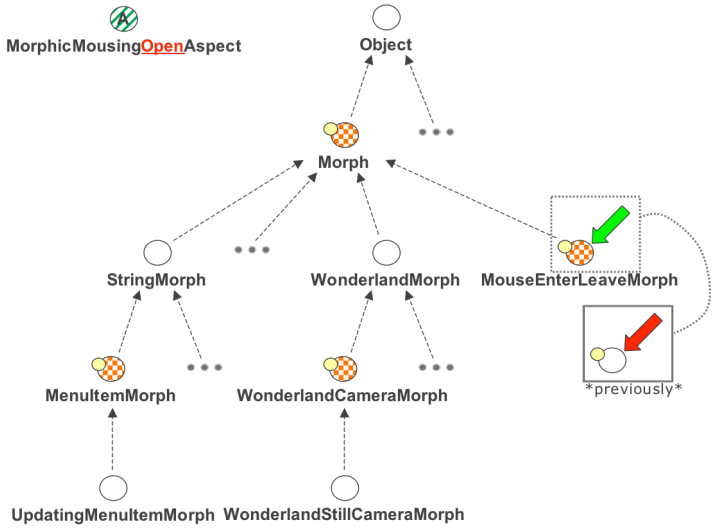

Fig. 21. Visualization of composition after class addition

```
MouseEnterLeaveMorph removeSelector: #handlesMouseOver:.
MouseEnterLeaveMorph removeSelector: #mouseEnter:.
MouseEnterLeaveMorph removeSelector: #mouseLeave:.

WonderlandCameraMorph removeSelector: #handlesMouseOver:.
WonderlandCameraMorph removeSelector:   #mouseEnter:.
WonderlandCameraMorph removeSelector: #mouseLeave:.
```

Fig. 22. Code removing mouse enter and leave methods

In Fig. 22 we see code that removes the code to handle mouse-enter and mouse-leave events from the two classes MouseEnterLeaveMorph and WonderlandCameraMorph. The resulting composition that is based on our reinstall-advice adaptation policy is illustrated in Fig. 23.

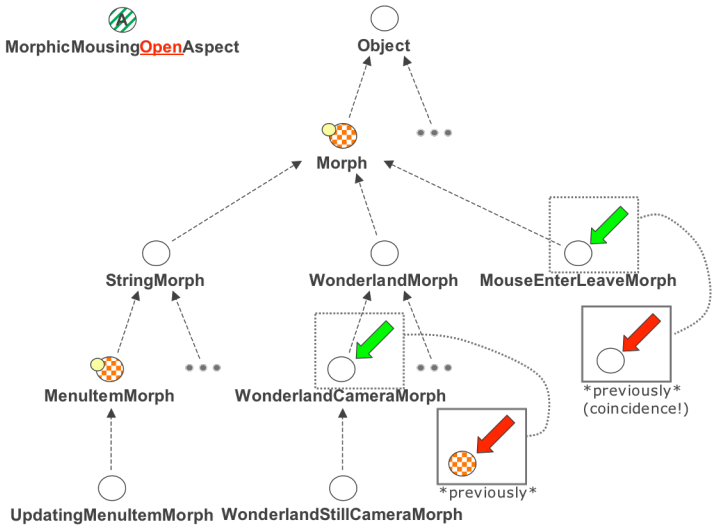

Fig. 23. Visualization of composition after method removal

Because of the removal the set of join-point shadows to be instrumented by the installation of the MorphicMousingOpenAspect has changed. This change caused our adaptation strategy to reinstall our aspect that now does not affect the aforementioned join-point shadows anymore.

6.3. Changing an Advice's Pointcut

Changing an advice's pointcut expression while instances of such aspect are active might also need to be addressed on a case-by-case basis. If deliberately ignored or accidentally overlooked, the example of a changed pointcut, as listed in Fig. 7, can leave our system in a state displayed in Fig. 8.

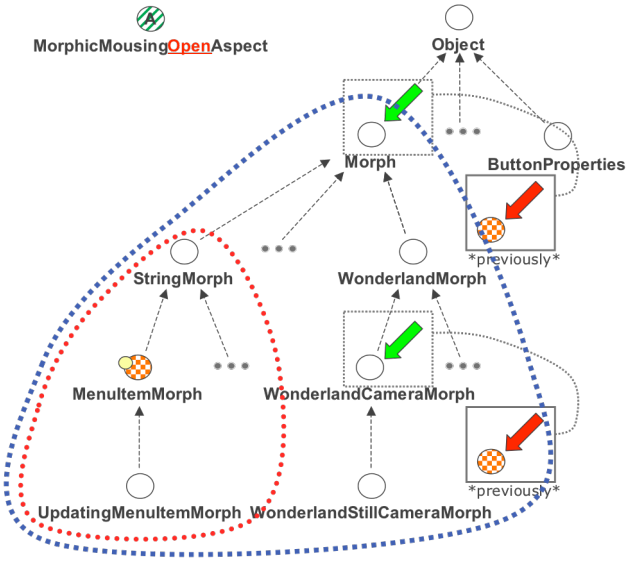

Fig. 24. Visualization of composition after change to pointcut coverage

The selection of an indifferent adaptation strategy for an advice of an open aspect would have lead to the same result. An adaptation strategy to reinstall all affected pieces of advice or the aspect they are associated with yields the change of aspect composition as illustrated in Fig. 24: Since mouseEnter: and mouseLeave: of Morph and WonderlandCameraMorph are not covered by the new version of our pointcut expression anymore (as indicated by the missing dots that have marked them previously), our aspect composition is revoked from there also.

6.4. Changing an Aspect's Advice

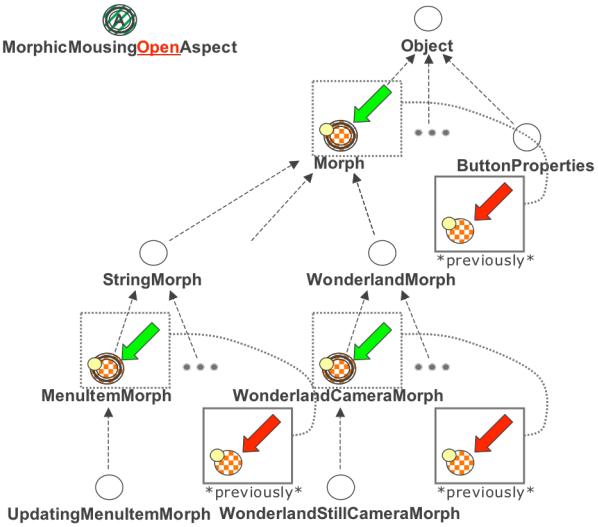

Fig. 25. Visualization of composition after change to advice

Changing the advice code of a composed aspect might cause defective system behavior as well. If deliberately ignored or accidentally overlooked, the example of changed advice code, as listed in Fig. 9, can leave our system in a state displayed in Fig. 10. The selection of an indifferent adaptation strategy for an advice of an open aspect would have lead to the same result.

An adaptation strategy to reinstall all affected pieces of advice or the aspect they are associated with yields the change of aspect composition as illustrated in Fig. 25. Since the before blocks of adviceMouseEnter and adviceMouseLeave were changed, the composition of the two pieces of advice were adjusted as well to the new behavior (as indicated by the different style of the border of the class symbols the composition is made effective).

7. Related Work

7.1. Load time weaving

Class loading in Java represents a simple form of open systems. Hence, if a system is designed in a way that it uses classes that are not known at compile-time then the underlying system is open. JMangler [18], Javassist [6], and EuLisp [5] are systems that permit the adaptation of classes at runtime. Hence, it permits one to adapt system changes (namely the addition of new classes to a running system). From the technical perspective, these systems would allow to build up an adaptation model based on the loading-a-new-class system event. However, the possible actions of the adaptation model are quite restricted due to the underlying language design: while the adaptation of the newly loaded classes can be easily achieved, the deletion of adapted join point shadows is not possible for classes which are already instantiated.

7.2. Dynamic Weaving via Interpretation

There are already a number of systems that permit one to weave aspects dynamically based on the interpretation of the underlying code base. The most often mentioned system is PROSE [23,24], which is based on the programming language Java. It extends the underlying runtime system to invoke aspect-specific code when a certain join-point is reached by utilizing break points of the Java debugging interface. On an abstract level, this means that specific events (for example method or constructor calls) are redirected: for a given call it is checked to see whether aspect-specific code needs to be executed. From that point of view, they potentially permit weaving to react on changes of the underlying system: changes of the base system could be noted by having a corresponding join-point on the method that is responsible for the change. Due to the limitations of Java in respect to changes of the underlying system, however, the problems that typically arise in open systems rarely occur here: classes or single methods are not (or rarely) considered to change at runtime. In contrast to static weaving classes that were not present at compile time, but which are dynamically loaded, may also be considered by woven aspects. An explicit adaptation model as proposed in this paper is not considered.

7.3. Dynamic Weaving via Adaptation

A number of systems achieve dynamic weaving via code adaptation. Systems like Steamloom [3], AspectS, or the Selective Just-in-time Weaver [25] also permit dynamic weaving, but in contrast to the previously mentioned approach they adapt the underlying code base. Although they technically work quite differently, they have one point in common: at one point in time (when an aspect is woven) there is a computation that determines all join-point shadows [20] of a certain aspect and adapts these join-point shadows in some way. Loading a new class to the system differs from the previous approach. Loading a new class that was not available when the pointcut computation was done would not be considered by the corresponding aspects. Hence, such systems also suffer from the problem of open systems.

In AspectL [8], an aspect-oriented extension of CLOS [9], the concept of generic pointcuts is introduced that allow adding methods on the one hand and aspect weavers on the other hand. Whenever a method is added to a generic pointcut, all aspect weavers are triggered to generate new corresponding before/after/around methods. Likewise, whenever a new aspect weaver is added, it is applied to each already existing method in that pointcut. In AspectL, aspect weavers are not declarative but operational, making it necessary to use functions of the CLOS MOP [17]. AspectL does not provide a declarative pointcut language. In other words, AspectL provides some basic machinery to keep aspects and base code in sync, which can be understood as a first step towards Open Aspects, but does not provide a full-fledged solution as described in this paper.

In AspectS weaving is achieved by instrumentation of instances residing in the Smalltalk meta-object structure. Since this structure changes permanently during development and runtime, the problem addressed by Open Aspects occurs frequently. However, previous versions of AspectS did not take these problems into account.

7.4. Continuous Weaving

Morphing aspects as described in [11] permit one to weave aspects depending on the application's behavior. In particular, an aspect does not adapt all join-point shadows [20] where aspect-specific behavior is potentially needed upfront, but adapts a minimal set of join-point shadows. Additionally, it provides a computation strategy that describes how and when additional join-point shadows need to be adapted. As a consequence, weaving is not performed in a single step, but continuously during an application's runtime. However, the problem of changes in an open system is not addressed by morphing aspects: the adapted join-points where additional weaving needs to be performed are join-points of the underlying application, and not join-points of the underlying system. For example, the creation or deletion of a class is not considered as a join-point of Morphing Aspects in [11].

7.5. Incremental Weaving

As a general-purpose AOP extension to IBM VisualAge for Smalltalk, Apostle [1,2] offers an incremental weaving mechanism to make the development of aspects fit better into the interactive and exploratory development approach of Smalltalk. It allows code to be added or modified over time. Changes affecting aspect compositions cause the weaver to incrementally adjust these compositions. Its incremental weaver, however, does not allow one to modify its behavior but reacts to changes always the same way.

7.6. Consistency Maintenance

SmartTalk [21] implements the ability to keep the evolution of classes consistent with specified contracts related to subclassing semantics, base module properties, and the protection of implementation internals. In SmartTalk class changes are monitored and potential conflict situations such as accidental method capture or inconsistent methods are detected. Then, classes responsible for a particular conflict situation are informed, and automatic transformations handling such conflict are processed. Similar to Open Aspects, this approach is concerned with consistency in an open environment. Both SmartTalk and Open Aspects address changes to the underlying system in order to deal with inconsistencies. While SmartTalk is only concerned with changes to the bases system but not the transformations, Open Aspects deals with changes to both the base and the transformation system.

8. Summary and Conclusion

In this paper we identify the need to handle the weaving of aspects in open systems, that is to say systems that are intentionally designed to change at runtime, in a special way: it is necessary that the developer specifies how aspect compositions are to be maintained if changes to the system might affect them.

Open Aspects allow a system to respond to system change events appropriately by providing adaptation models. Adaptation models explicitly associate such change events with corrective actions to be taken in the event of change – change to the base system or to aspect compositions themselves. Additive, transformative, and subtractive changes to regular objects and aspects with their methods, fields, pieces of advice, and pointcuts are dealt with based on dynamic conditional weaving.

Open Aspects can be applied by developers and maintainers to make changes and their effects to pointcuts, aspects, pieces of advice, and the compositions' base system visible. They help to indicate inconsistencies between intended and actual compositions over time and provide means to take corrective actions if desired and necessary.

OpenAspectS is our implementation of Open Aspects in Squeak/Smalltalk. It extends AspectS' runtime structure to become aware of system changes and to perform relevance checks upon which it is decided if a particular change needs to be addressed by an adaptation strategy or not at all.

9. Acknowledgements

We would like to thank Alexandre Bergel, Gilad Bracha, Pascal Costanza, Stephane Ducasse, Jeff Eastman, Erik Ernst, and Dave Thomas for their valuable discussions and contributions.

References

[1] B. de Alwis. *Aspects of Incremental Programming.* Master's Thesis, University of British Columbia, Vancouver, Canada, 2002.
[2] B. de Alwis and G. Kiczales. *Apostle: A Simple Incremental Weaver for a Dynamic Aspect Language.* Technical Report TR-2003-16, Department of Computer Science, University of British Columbia, Vancouver, Canada, 2003.
[3] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD),* pp. 83-92, ACM Press, Lancaster, UK, March 22-26, 2004.
[4] J. Brant, B. Foote, R.E. Johnson, and D. Roberts. Wrappers to the Rescue, In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP),* pp. 396-417, LNCS 1445, Springer, 1998.
[5] H. Bretthauer and J. Kopp. Balancing the EuLisp Metaobject Protocol. In *Proceedings of the International Workshop on New Models for Software Architecture,* Tokyo, Japan, 1992.
[6] S. Chiba. Load-time Structural Reflection in Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP),* pp. 313-336, LNCS 1850, Springer, 2000.
[7] ChiMu Corporation. Java *and Smalltalk syntax compared.* (http://www.chimu.com/publications/JavaSmalltalkSyntax.html) 2000.
[8] P. Costanza. A Short Overview of AspectL. In *Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS),* Berlin, Germany, September 23-24, 2004.
[9] R. Gabriel, J. White, and D. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. Communications of the ACM, 34 (9), 1991, pp. 28-38.
[10] A. Goldberg and D. Robson. *Smalltalk 80 – The Language and its Implementation.* Addison-Wesley, 1983.
[11] S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD),* pp. 46-55, ACM Press, Lancaster, UK, March 22-26, 2004.
[12] R. Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. In *M. Aksit, M. Mezini, R. Unland, editors, Objects, Components, Architectures, Services, and Applications for a Networked World,* pp. 216-232, LNCS 2591, Springer, 2003.
[13] R. Hirschfeld and K. Kawamura. Dynamic Service Adaptation. In *Proceedings of the ICDCS 2004 Workshop on Distributed Auto-adaptive and Reconfigurable Systems (DARES),* pp. 290-297, IEEE Press, Tokyo, Japan, March 23-24, 2004.
[14] R. Hirschfeld, K. Kawamura, and H. Berndt. Dynamic Service Adaptation for Runtime System Extensions. In *R. Battiti, R. lo Cigno, M. Conti, editors, Wireless On-Demand Network Systems, Proceedings of WONS2004,* LNCS 2928, pp. 225-238, Springer 2004.
[15] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA),* pp. 318-326, ACM Press, Atlanta, GA, USA, October 5-9, 1997.
[16] K. Kawamura, J. Hamard, R. Hirschfeld, and A. Minokuchi, and B. Souville. Sustainable Evolutionary Systems. In *NTT DoCoMo Technical Journal,* vol. 6, no. 1, pp. 14-19, June 2004 (Japanese version appeared in NTT DoCoMo Technical Journal, vol. 12, no. 1, pp. 15-19, April 2004).
[17] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol.* Addison-Wesley, 1991.
[18] G. Kniesel, P. Costanza, and M. Austermann. JMangler – A Powerful Back-End for Aspect-Oriented Programming, In *R. Filman, T. Elrad, D. Clarke, M. Aksit (eds.). Aspect-Oriented Software Development,* Prentice Hall, 2004.
[19] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD),* pp. 41-55, ACM Press, Enschede, The Netherlands, April 22-26, 2002.
[20] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs, In *Proceedings of Compiler Construction (CC),* pp. 46-60, LNCS 2622, Springer, 2003.
[21] M. Mezini. Maintaining the Consistency of Class Libraries During Their Evolution. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA),* pp. 1-22, ACM Press, Atlanta, GA, USA, October 5-9, 1997.
[22] R. Pawlack, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Proceedings of Reflection 2001,* pp 1-24, LNCS 2192, Springer, 2001.
[23] A. Popovici, Th. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD),* pp. 141-147, ACM Press, Enschede, The Netherlands, April 22-26, 2002.
[24] A. Popovici, Th. Gross, and G. Alonso. Just in Time Aspects. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD),* pp.100-109, ACM Press, Boston, MA, United States, March 17-21, 2003.
[25] Y. Sato, S. Chiba, and M. Tatsubori. A Selective, Just-In-Time Aspect Weaver, In *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE),* pp. 189-208, LNCS 2830, Springer, 2003.

Towards
Unified Aspect-Oriented Programming

Noury Bouraqadi Abdelhak Seriai Gabriel Leblanc

{*bouraqadi, seriai*}*@ensm-douai.fr*
École des Mines de Douai - Dépt. G.I.P.
941, rue Charles Bourseul - B.P. 10838
59508 Douai Cedex - France

Abstract

Aspect-Oriented Programming (AOP) is a paradigm that aims at improving software modularization. Indeed, aspects are yet another dimension for structuring applications. The notion of aspect refers to any crosscutting property. Such crosscutting can be either dynamic or static. Dynamic crosscutting refers to applications execution flow. While, static crosscutting refers to applications structure. Although many AOP approaches does enable these two kinds of crosscutting, this support is not always satisfactory. Aspects code is complex and often requires different constructs for expressing static and dynamic crosscutting. We present in this paper the foundation for an AOP platform that unifies the description of both kinds of crosscuttings. This solution relies on reflection and mixin-based inheritance.

Key words: aspect-oriented programming, static crosscutting, dynamic crosscutting, reflection, mixin-based inheritance

1 Introduction

Aspect-Oriented Programming (AOP) [15,10,11] is among key post-object paradigms that appeared during the last decade. This programming approach supports separation of concerns. Building an application using the AOP approach leads to defining on the one hand one *application core*, and on the other hand an arbitrary number of *aspects*. Application core is usually a set of classes. Aspects are concerns that crosscut application core. Aspects are not only separated from application core, they are also isolated one from the other. Hence, AOP promotes modularization. Development responsibilities of aspects and application core can be dispatched among members of a project team. Once all modules (aspects and application core) are ready, the full application can be "integrated" through the process of *weaving*.

The notion of aspect refers to any property crosscutting a software. Such crosscutting can be either *dynamic* or *static* [16].

Dynamic crosscutting: A crosscutting is said to be dynamic if it affects applications behavior, *i.e.* execution flow. The implementation of dynamic crosscuttings relies on the concepts of *pointcuts* (set of points within the execution flow) and *advices* (blocks of code to evaluate at some points of the execution flow).

Static crosscutting: A crosscutting is said to be static if it affects applications structure. The implementation of static crosscuttings relies on *introductions* of new building blocks (*e.g.* classes, methods, instance variables) and restructuring their relationships (*e.g.* inheritance).

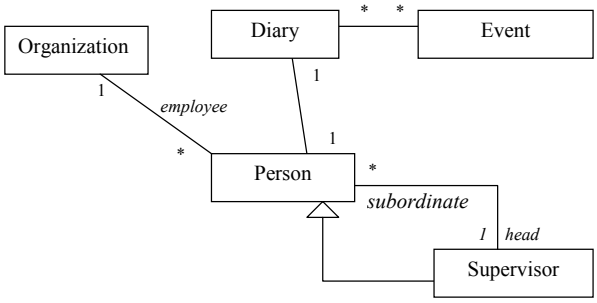

Fig. 1. Distributed Diary System Core

We illustrate these two of aspects using an example of a distributed diary system. Application core for this system is a set of classes describing employees, organization, diaries and events (see figure 1). This system can have different aspects. We present in the following two of them: log which illustrate dynamic crosscutting and absence management which illustrate static crosscutting.

The log aspect displays on a console traces describing system's activity. So, logs can be produced on the addition a new employee to the organization or events addition/removal to/from diaries. Log is a typical aspect with dynamic crosscutting. Trace production is triggered by the application execution. If no execution happens, the log aspect computations (*i.e.* logging) isn't performed. This is often the case of "infrastructure oriented" aspects.

The absence management aspect deals with employees vacations. Each person has a certain amount of available vacation days and can request vacations. Vacation requests has to be validated by the requester's boss before a new event is added to the requester's diary. Storing available vacation days requires a new instance variable to be inserted into the Person class. Handling requests and updating available vacation days count requires new methods to be inserted in classes Person and Supervisor. Absence management is a typical aspect with static crosscutting. It extends existing classes with new instance variables and methods. This is often the case of "business oriented" aspects.

Among existing AOP approaches, some does support only one kind of crosscut-

ting [7,2,8,9,19]. Others [16,13,21] do support both static and dynamic crosscuttings. But, these platforms lead quickly to complex code, even for simple aspects. And, often aspects definitions are non uniform: different constructs are used for expressing code introductions and advices. This is particularly true for AspectJ the most popular AOP language, as we show in section 2.1.

In this paper, we setup the foundations for a platform that unifies the programming of crosscuttings should they be static or dynamic. We use *reflection* [23,18] and *mixin-based inheritance* [6] to extend an object-oriented platform in order to supports AOP. No new language construct is needed, and only a minimal set of concepts is introduced and applied uniformly to express both static and dynamic crosscutting. We believe that this platform provides simplicity and uniformity, without scarifying expressiveness. Implementations of simple aspects remain simple, and those of complex aspects are still possible.

Reflection is the ability of a system to reason on and to act upon itself. In the context of programming languages, reflection provides developers with two programming levels: a *base-level* and a *meta-level*. The base-level is where applications building blocks (*i.e.* structure) are defined. The meta-level is where applications semantics (*i.e.* behavior) is defined. Programming within both base- and meta-levels is uniform since it relies on the same constructs in a reflective language.

Having access to applications structure and behavior is not enough to define aspects. Applications need to be decomposed so that each aspect definition is isolated and separated from the others. We use mixin-based inheritance to achieve this separation. Mixin-based inheritance is an alternative to multiple-inheritance which avoids automatic linearization issues. A mixin can be viewed as a subclass parametrizable with its superclass. In the proposed solution, each aspect is defined as a set of mixins. This description applies uniformly to express both static and dynamic crosscuttings.

With our model, each class of the application core is linked to a meta-level class. The process of weaving inserts mixins into class hierarchies. Mixins related to static crosscuttings are inserted into the hierarchy of base-level classes. While, mixins related to dynamic crosscuttings are inserted into the hierarchy of meta-level classes.

The remainder of this paper is organized as follows. Section 2 motivates the need for a platform supporting the definition of both functional and non-functional aspects. This motivation is illustrated using a distributed diary example that will be used throughout the paper. Then, foundations of unified AOP based on reflection and mixin-based inheritance is described in section 3. Last, after discussion related work in section 5, section 6 ends the article ends with concluding remarks and some perspectives.

2 Motivation

In this section, we motivate the need of a unified AOP based on AspectJ, the AOP mainstream platform. This motivation is illustrated using the distributed diary example exposed in the introduction. We show some limitations of AspectJ when it comes to building reusable aspects with static and dynamic crosscutings, namely the log aspect and the absence management aspect.

2.1 Some AspectJ Limitations

2.1.1 A First Absence Management Aspect

```
01: public aspect SimpleAbsenceManagement {
02:  private int Person.vacationDaysCount;

03:  public int Person.getVacationDaysCount(){
04:    return this.vacationDaysCount;}

05:  public void Person.setVacationDaysCount(int newVacationDaysCount){
06:    this.vacationDaysCount = newVacationDaysCount;}

07:  public String Person.toString(){
08:    return "\nAvailable Vacation Days = " + this.getVacationDaysCount();}

09:  public int Person.defaultVacationDaysCount(){
10:    return 30;}

11:  pointcut constructorExec(Person aPerson):
12:    execution(Person.new(String, String)) && target(aPerson);

13:  after(Person aPerson): constructorExec(aPerson){
14:    aPerson.setVacationDaysCount(aPerson.defaultVacationDaysCount());}

...
```

Fig. 2. A simple implementation of the absence management aspect in AspectJ

Figure 2 gives a first version [1] of the absence management aspect in AspectJ. This aspect is not reusable since it directly refers to application core class Person. Indeed, a new field named vacationDaysCount is introduced in class Person for counting available vacation days. This vacation days counter is used for vacation requests (not shown on figure 2) and also for the string describing person returned by the toString() method.

[1] We provide here only part of the actual code of the aspect.

In order to initialize the counter, the only possibility is to use an advice (lines 13 and 14) that acts after the execution of the constructor of class Person (lines 11 and 12). Figure 3 provides a simple program using the classes Person and Supervisor after weaving the SimpleAbsenceManagement aspect.

Evaluated Code

```
Supervisor chief = new Supervisor("Bart", "Simpson");
Person joe = new Person("Joe", "Dalton");
joe.setBoss(chief);
System.out.println("---println(joe)---\n" + joe);
System.out.println("---println(chief)---\n" + chief);
```

Console Display

```
---println(joe)----
Available Vacation Days = 30
---println(chief)---
Available Vacation Days = 30
Boss of 1 person(s)
```

Fig. 3. A code evaluated and its result after weaving the SimpleAbsenceManagement aspect

In this example, we can see different limitations of AspectJ. First, a simple extension of an existing code can lead to somewhat complex code. This is the case with the initialization of the vacationDaysCount field. Such initialization which simply requires extending an existing constructor is actually performed using a rather "unnatural" code based on a pointcut and an advice.

Another problem can arise on evolution. Suppose we add a toString() method into the Person class definition. In this case, AspectJ fails weaving the SimpleAbsenceManagement aspect and reports a conflict. The only solution to this conflict is to replace the defintion of toString() method provided by the aspect with pointcut and advice constructs (see figure 4). Note that the within(Person) condition in the pointcut description ensures that the advice is performed only once: for the toString() method defined within the Person class. Otherwise, the advice would be performed twice for instances of the Supervisor class, since this latter does redefine the toString() method and does and super send.

```
1: pointcut toStringExec(Person aPerson):
2:  execution(String Person.toString()) && target(aPerson) && within(Person);

3: String around(Person aPerson): toStringExec(aPerson){
4:  String initialString = proceed(aPerson);
5:  return initialString +
6:    "\nAvailable Vacation Days = " + aPerson.getVacationDaysCount();}
```

Fig. 4. Replacement of the toString() method definition with a pointcut and an advice in the SimpleAbsenceManagement aspect

A similar problem arise when two aspects introduce methods with the same signature in the same class. In this case, weaving fails and one needs to rewrite at least one of the two aspects and replace method introduction with statements based on the pointcut and advice constructs.

Note that AspectJ weaver does handle the case of homonymous fields. Fields scopes are restricted to aspects where they are defined. For example, let A1 and A2 two aspects that introduce within the same class fields of the same name. Methods introduced in A1 will access the field introduced in A1. And methods introduced in A2 will access the field introduced in A2. The same solution applies if an aspect introduces a field with a name already used in core application code. Although this solution is convenient for most cases, sometimes one may want to merge such fields in order to share data.

Last, we can note that using AspectJ one can easily end up "hardwiring" aspect definitions to a particular application. This is the case of the SimpleAbsenceManagement which explicitly refer to the Person class. In the following, we'll see that disciplined programming can avoid this pitfall and enable aspect reuse. However, we'll face other limitations.

2.1.2 A Reusbale Absence Management Aspect

Figure 5 provides the definition of a reusable absence management aspect [2] in AspectJ. Actually, there are two aspects. The first one AbsenceManagement (lines 1 to 15) is reusable cause not bound to any application code. The second aspect AbsenceManagementImpl (lines 16 and 17) extends the former with links to the core application code.

The AbsenceManagement introduces an new "marker" interface named AbsenceRequestor (line 2). All classes which implement this interface will be extended with members introduced in lines 3 to 10. The actual class which is extended this way is Person which is referenced in AbsenceManagementImpl (line 17). The Person class is linked to the marker interface AbsenceRequestor using the "declare parents" statement.

The use of a marker interface has a consequence on the pointcut declaration which enables the initialization of the vacationDaysCount field (lines 11 to 13). Because interfaces does not hold constructors, the execution statement should refer to constructors of classes implementing the interface. This is what the "+" refers to in the expression execution(AbsenceRequestor+.new(..)). However, this definition covers not only classes directly implementing the interface (Person in our example), but also their subclasses (Supervisor in our example). In order to avoid executing the advice twice, we need to complexify a bit more the pointcut decla-

[2] We provide here only part of the actual code of the aspect.

```
01: public abstract aspect AbsenceManagement {

02:  public interface AbsenceRequestor {}

03:  private int AbsenceRequestor.vacationDaysCount;

04:  public int AbsenceRequestor.getVacationDaysCount(){
05:    return this.vacationDaysCount;}

06:  public void AbsenceRequestor.setVacationDaysCount(int newCount){
07:    this.vacationDaysCount = newCount;}

08:  public int AbsenceRequestor.defaultVacationDaysCount(){return 30;}

09:  public String AbsenceRequestor.toString(){
10:    return "Available Vacation Days = " + this.getVacationDaysCount();}

11:  pointcut constructorExec(AbsenceRequestor requestor):
12:    execution(AbsenceRequestor+.new(..)) && target(requestor) &&
13:    !cflowbelow(execution(AbsenceRequestor+.new(..)));

14:  after(AbsenceRequestor requestor): constructorExec(requestor){
15:    requestor.setVacationDaysCount(requestor.defaultVacationDaysCount());}

                                    ...

16: public aspect AbsenceManagementImpl extends AbsenceManagement{
17:   declare parents : Person implements AbsenceRequestor;
                                    ...
```

Fig. 5. A reusable implementation of the absence management aspect in AspectJ

ration. This is what is stated by line 13. Note however, that we don't get exactly the behavior provided in figure 2 (page 30). Indeed, with the reusable definition of the absence management aspect (introduced in this section), the initialization for instances of class Supervisor (subclass of Person) is done after all constructors (defined in classes Supervisor and Person) are executed. While in the non-reusable definition of the aspect (introduced in section 2.1.1 page 30), the advice is done right after the execution of the constructor of class Person.

Another problem with the reusable definition of the AbsenceManagement aspect provided on figure 5 is caused by the introduction of new methods such as toString() (lines 9 and 10). This extension performs well if the Person class does not implement a method with the same signature. However, if Person does implement a such method, than the extension is simply ignored without warning. Similarly two aspects introducing in a same class two methods with the same signature, the weaver does actually silently introduce only one methods without warning. Even if warnings were available, aspect integrators would have to change the

aspects definitions and hence loose part of the benefice of reuse.

```
1: pointcut toStringExec(AbsenceRequestor requestor):
2:  execution(String AbsenceRequestor.toString()) &&
3:  target(requestor) &&
4:  !cflowbelow(execution(String AbsenceRequestor.toString()));

5: String around(AbsenceRequestor requestor): toStringExec(requestor){
6:  String initialAnswer = proceed(requestor);
7:    return initialAnswer + "\nAvailable Vacation Days = "
8:      + requestor.getVacationDaysCount();}

//Default method
9: public String AbsenceRequestor.toString(){return "";}
```

Fig. 6. Replacement of the toString() method definition with a pointcut and an advice in the AbsenceManagement aspect

To avoid such problems, one should replace every method with pointcuts and advices. Figure 6 provides such rewriting for the toString() method. The pointcut declaration captures the execution of method toString() by instances of classes implementing the AbsenceRequestor interface. The !cflowbelow(...) part of the declaration avoids performing the advice twice when there are super.toString() sends. However, the resulting semantics is a bit different from the one obtained with the non-reusable version of the aspect (Figure 2 page 30). As shown by figure 7, the string corresponding to available vacation days is appended at the end of supervisors descriptions. While in the non-reusable aspect definition (Figure 3 page 31) available vacation days string is inserted before the string providing the number of subordinates.

Evaluated Code

```
Supervisor chief = new Supervisor("Bart", "Simpson");
Person joe = new Person("Joe", "Dalton");
joe.setBoss(chief);
System.out.println("---println(joe)---\n" + joe);
System.out.println("---println(chief)---\n" + chief);
```

Console Display

```
---println(joe)----
Available Vacation Days = 30
---println(chief)---
Boss of 1 person(s)
Available Vacation Days = 30
```

Fig. 7. A code evaluated and its result in the context of the reusable absence management aspect

Yet another problem with AspectJ is that the code provided by figure 6 (page 34) need to insert a *default* implementation of the toString() method (line 9). This definition is useful for cases where the core application classes does not provide such

a method. When such method is available, the default implementation is simply ignored. While the use of a default method implementation allows reusing the aspect in multiple applications providing or not the introduced method, it causes a non resolvable conflict when two aspects provide two default implementations of the same method. Indeed, the default method implementation is just a programming style and the weaver is not aware of it. So, weaving fails, and application integrators has to change one aspect and remove the corresponding default method implementation.

2.1.3 Summary of AspectJ Limitations

To sum up, AspectJ has several limitations regarding a uniform description of reusable aspects.

- AspectJ does not encourage reuse. Building reusable aspects mainly relies on developers discipline.
- AspectJ introduces extra complexity for developers. They are offered two different sets of constructs for implementing cross-cutting code: inter-type declarations for static cross-cutting, and pointcuts and advices for dynamic cross-cutting.
- Reusable definitions of simple aspects is complex and unatural, since it requires having for each introduced method an inter-type declaration with the default implementation of the method, a pointcut declaration capturing a single execution of the method and an advice performing the desired processing.
- It is difficult if not impossible to always get the desired semantics when building reusable aspects.
- It is not possible to build fully reusable aspect. Application integrator may always face conflicts requiring modifying aspects code.

2.2 Problem Statement

Starting from AspectJ limitations, we list here issues that should be addressed by an AOP platform allowing to build reusable aspects with both static and dynamic crosscutting. Such platform should be easy to learn and use, especially when it comes to maintain existing aspects. As proved by languages such as Self [24] and Smalltalk [12], uniformity and simplicity can go along with the language expressive power. We believe that this philosophy should be adopted in AOP platforms. Issues to be addressed are the following:

Reusable Aspects: An AOP platform should encourage building reusable aspects, by encouraging decoupling aspect's code from other applications parts.

Uniform Description of Crosscuttings: Having a small set of constructs uniformly used to express both static and dynamic crosscuttings would ease the learning

and the understanding of aspects.

Uniform Conflict Management: Conflicts can occur between two static crosscuttings or two dynamic ones alike. Developers should be provided the same tools to handle both of them.

Crosscuttings Interactions: Dynamic crosscuttings should be able to alter the whole application code including static crosscuttings.

3 Foundations for Unified AOP

Our proposal to support unified AOP relies on *reflection* [23,18] and *mixin-based inheritance* [6]. Starting from plain Smalltalk, we introduce a minimal set of concepts and apply them uniformly to express both static and dynamic crosscutting. We believe that with this platform provides simplicity and uniformity, without scarifying expressiveness.

In this section, we first briefly remind reflection and mixin-based inheritance. We then provide a description of aspects in a platform supporting unified AOP. Last we describe the process of weaving aspects into application core.

3.1 Background: Reflection and Mixin-Based Inheritance

3.1.1 Reflection

Refection is the ability of a system to reason and to act upon itself. In the context of object-oriented languages, reflection gives access to languages semantics. A reflective OO language provides programmers with two programming levels: *base-level* and *meta-level*. The base-level includes all application objects (e.g. diary, person, supervisor, ...). The meta-level includes so-called *meta-objects* which are objects describing the reflective language's constructs (e.g. classes) and how programs are evaluated (e.g. message dispatch).

We use in the reminder of this paper the Meta-Object Protocol (MOP) of MetaclassTalk [3] [5,4], a reflective extension of Smalltalk. MetaclassTalk MOP allows controlling objects creation and memory allocation, instance variable reads and writes, message sends and receptions, and method lookup and evaluation.

[3] http://csl.ensm-douai.fr/MetaclassTalk

3.1.2 Mixins

The concept of mixins has been introduced as an alternative to both single and multiple inheritance. It provides more code sharing than allowed with single inheritance, while avoiding issues arising with multiple inheritance and its automatic linearization. A mixin can be viewed as an abstract subclass parameterized with its superclass. This parameterization allow using a same mixin in different class hierarchies.

The mixin model we use in the reminder of this paper is inspired by the one introduced in CLOS [14]. A class can have many superclasses (mixins or plain classes). But, we go further than CLOS where mixin-based inheritance is just a programming style. We constrain the model to allow only multiple inheritance of mixins [3,4]. A subclass can inherit from an arbitrary number of mixins, but can have only one non-mixin superclass. Linearization chain of a subclass starts with mixins in the order provided in the subclass definition. The non-mixin superclass appears after mixins. So, methods are looked up first in mixins and then in the non-mixin superclass.

3.2 Structure of Unified Aspects

Our proposal relies on using mixins to build unified aspects. A unified aspect is a compound of: a Set of mixins, a pre-weaving script, and a post-weaving script. Mixins provide descriptions of crosscutting code. While, pre-weaving and post-weaving scripts are sequences of Smalltalk statements describing initialization operations to be performed before and after weaving crosscutting code into application core.

A crosscutting be it static or dynamic is implemented using a set of mixins. Mixins describing static crosscuttings are aimed to be inserted (on weave-time) into base-level class hierarchies. While, mixins describing dynamic corsscuttings are aimed to be inserted (on weave-time) into meta-level class hierarchies.

Besides aspects, developers have also to provide application core. That is a set of classes organized using composition and inheritance relationships. These classes define basic structure and behavior of application objects. They do not hold any instance variable or method related to any aspect.

3.3 Weaving Unified Aspects

As mentioned before, weaving relies on reflection, and mixin-based inheritance. For every class A in application core, the weaver builds a meta-object class AMeta. Each instance of A is controlled by an instance of AMeta. So, instances of AMeta

provide the semantics of behavior (message sends and receptions) and structure (instance variables reads and writes) of instances of A. An application is obtained once mixins provided by various aspects are linked through inheritance to application core classes and corresponding meta-object classes. It worth noting that the meta-object class hierarchy is parallel to the class one. So, given B a subclass of A, BMeta the class of meta-objects of instances of B is built by the weaver as a subclass of AMeta

Once the weaver creates meta-object classes, it repeats the four following steps for every aspect.

(1) Provide a map stating which aspect's mixins to link to which core application classes. This step is necessary because the aspects' definitions does not refer to application core classes.
(2) Evaluate the current aspect's pre-weaving script.
(3) Link the current aspect's mixins to application core classes.
(4) Evaluate the current aspect's post-weaving script.

Because of the use of mixin-based inheritance, the cross-cutting code remains isolated from application core (though it is linked). Relying on Smalltalk dynamicity our solution supports not only dynamic weaving, but also dynamic unweaving. To complete this support, the structure of unified aspects includes also pre-weaving and post-weaving scripts. The pre-weaving script is evaluated before unlinking classes and mixins. The post-weaving script is evaluated after unlinking classes and mixins.

Joint point are expressed using the map and the MOP. That is, joint point cover class definitions, message sends and receptions, and instance variables reads and writes.

3.4 Aspects Interactions and Conflicts

Aspects does not only alter the structure and behavior of the application core, they also may affects each other execution. Consider an aspect A1 that makes a core application class C inherit from some mixin M1. Suppose also that we weave into this application another aspect A2 that adds some other mixin M2 to the superclass list of class CMetaObject, the class of meta-objects of instances of class C. Therefore, the semantics of code introduced by the A1 aspect using the M1 mixin is altered by the A2 aspect which introduces the M2 mixin.

Conflicts may arise when two aspects link mixins with homonymous methods or instance variables to a same class. Mixin-based inheritance provides us with a first solution to this open issue. Indeed, developers can order mixins linked to each class. Methods introduced in mixins appearing first in a class definition override homony-

mous methods defined in other mixins. This solution currently implemented in our prototype is rather coarse grain and does not address the case of homonymous instance variables. Van Limberghen and Mens [17] describe a solution that tackles this problem.

Yet another cause of conflicts is having weaving scripts of different aspects perform "contradictory" actions (e.g. setting some class variable to different values). We address this issue by allowing developers choose aspects precedence as in AspectJ. That is, developers choose the order of weaving. However, the resolution of this kind of conflicts deserves further investigations we defer to future work.

4 Examples

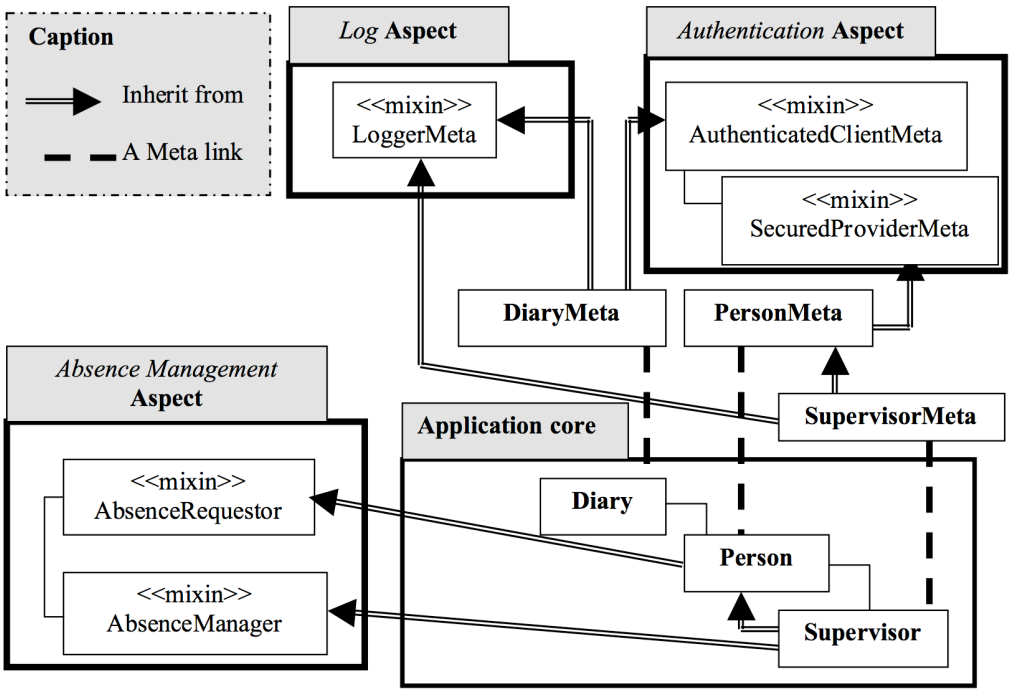

Fig. 8. A Subset of the distributed diary system built with our solution

Figure 8 shows part of the distributed diary system built using our solution after weaving [4]. Application core includes various classes: Person, Supervisor, and Diary. However, these classes does not define any instance variable or method related to aspects such as absence management, log or authentication.

[4] The full code is available on-line at http://csl.ensm-douai.fr/MetaclassTalk

The absence management aspect introduces two new roles: *absenceRequester* and *absenceManager*. An *absenceRequester* is supposed to store an available vacation days. It is also supposed to understand the requestVacation: message which argument is the vacation duration in days. As a response to this message an *absenceRequester* checks if the duration is less than or equal to available vacation days and then requests the confirmation of an *absenceManager* (message acceptVacation: *duration* for: *anAbsenceRequester*). When the *absenceManager* accepts the request, the *absenceRequester* decrements available vacation days counter and inserts an event describing the absence into a diary.

```
Aspect subclass: #AbsenceManagementAspect
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Unified AOP-Diary Example-Aspects'.

AbsenceManagementAspect >> initialize
super initialize.
self addAllMixins: {AbsenceRequester. AbsenceManager}
```

Fig. 9. Definition of the "absence management" aspect

Each one of the above described roles is implemented using a mixin. So, mixins AbsenceRequester and AbsenceManager define appropriate instance variables and methods for handling vacation requests. So, the description of the "absence management" aspect includes only these two mixins. Figure 9 shows that this description consist in defining a class which instances have two mixins: AbsenceRequester and AbsenceManager. Pre-weaving and post-weaving scripts are implemented as methods in the aspect's class. Because here we don't need any special processing, we don't override the existing empty implementations provided by class Aspect. It worth noting that there is no direct reference to application core. Therefore, this aspect can be reused in other applications.

```
| absenceAspect |
absenceAspect := AbsenceManagementAspect new.
absenceAspect map: AbsenceRequester to: Person.
absenceAspect map: AbsenceManager to: Supervisor.
absenceAspect weave.
```

Fig. 10. Weaving the "absence management" aspect

To weave the "absence management" into our application, we need first to map each of its mixins to application core classes. In our implementation, an aspect is but an object that can be parameterized with a map describing which mixins to link to which application core classes (see figure 10). Then, by sending the weave message to the aspect, the weaving is finished. First, pre-weaving script is performed.

Then, the AbsenceRequester mixin is added to the superclasses list of class Person. Next, the AbsenceManager mixin is added to the superclasses list of class Supervisor. Last, the post-weaving script is performed.

4.2 Example of dynamic cross-cutting: the authentication aspect

As mentionned above, an aspect definition can include mixins that can go either to the base-level or to the meta-level. The "absence management" presented in the previous subsection is an aspect which definition relies on mixins that are to be linked to base-level classes. Here we present the "authentication" aspect which implementation relies on changing the semantics of message dispatch. So, it defines mixins that are to be linked to meta-object classes.

The authentication aspect introduces two roles: *authenticatedClient* and *securedProvider*. An *authenticatedClient* holds a login and a password that grant him access to services of some *securedProvider*. So, when an *authenticatedClient* needs to send some message to a *securedProvider*, *authorizedClient* first sends the pair login and a password to the *serviceProvider*. A *securedProvider* accepts processing only messages sent by client with valid login and password.

The authentication aspect implements these two roles using two meta-level mixins AuthenticatedClientMeta and SecuredProviderMeta. Figure 11 provides the actual code of these two mixins. We can see that mixin AuthenticatedClientMeta extends the semantics of message sending. It overrides method send:from:to:arguments: introduced in the MetaclassTalk MOP to perform first authentication before actually sending messages. Mixin SecuredProviderMeta extends the semantics of message reception. It overrides method receive:from:to:arguments: introduced in the MetaclassTalk MOP to actually perform received message from only authenticated clients.

The obtained authentication aspect is reusable since it does not refer to any core application class. Now, let see how to weave it. In our diary application accesses to a given diary have to be restricted to only some persons (*e.g.* its owner). Hence, instances of Person should be authenticated before message sends to instances of Diary. And, instances of Diary should check authorizations on message receptions. To get this behavior, we map mixin AuthenticatedClientMeta to class PersonMeta and mixin SecuredProviderMeta to class DiaryMeta. After weaving we get the mixins and meta-object classes linked.

Figure 12 provides an example showing how actually authentication is performed. Every instance pers1 of class Person is linked to a meta-object pers1Meta instance of PersonMeta. And, every instance diary1 of class Diary is linked to a meta-object diary1Meta instance of DiaryMeta. When pers1 sends some message, say addEvent:, to diary1, the message sending is intercepted by the pers1Meta

```
Mixin named: #AuthenticatedClientMeta
  instanceVariables: 'login password '
  category: 'Unified AOP-Diary Example-Aspects'.

AuthenticatedClientMetat ≫ login: loginString password: passwordString
  login := loginString.
  password := passwordString

AuthenticatedClient ≫ send: selector from: sender to: receiver arguments:
args
  receiver metaObject authentify: sender login: login password: password.
  ↑super send: selector from: sender to: receiver arguments: args
```

```
Mixin named: #SecuredProviderMeta
  instanceVariables: 'passwordDict authenticatedClients'
  category: 'Unified AOP-Diary Example-Aspects'.

SecuredProviderMeta ≫ initialize
  super initialize.
  passwordDict := Dictionary new.
  authenticatedClients := Set new

SecuredProviderMeta ≫ authentify: client login: login password: tentativePass-
word
  | actualPassword |
  actualPassword := passwordDict at: login ifAbsent: [↑self].
  actualPassword = tentativePassword ifFalse: [↑self].
  authenticatedClients add: client

SecuredProviderMeta ≫ acceptMsg: selector from: sender to: receiver
  ↑sender == receiver or: [
    (authenticatedClients includes: sender)]

SecuredProviderMeta ≫ receive: selector from: sender to: receiver argu-
ments: args
  (self acceptMsg: selector from: sender to: receiver) ifFalse: [
    ↑self error: 'Access restricted'].
  ↑super receive: selector from: sender to: receiver arguments: args
```

Fig. 11. Mixins for the “authentication” aspect

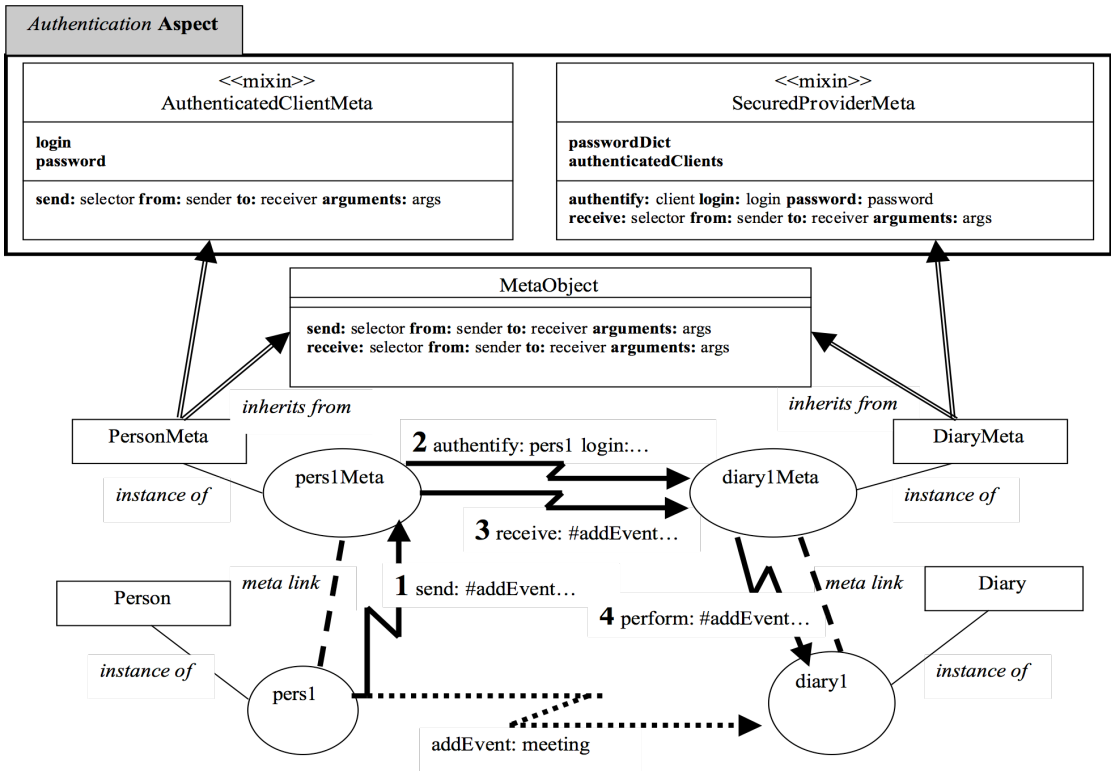

Fig. 12. Example of an authentication aspect in action

meta-object. This interception translates into a message send:. . . [5] implicitly dispatched (*i.e.* by the reflective infrastructure) to pers1Meta (step 1). Arguments of this message are informations about the addEvent: message (*e.g.* sender, receiver, selector,. . .). The meta-object pers1Meta attempts to do authentication by sending message authentify: *pers1* login: *loginOfPers1 password: passwordOfPers1* to diary1Meta, the meta-object of the receiver (step 2). Then, pers1Meta delivers the addEvent: message to perform to diary1Meta (step 3). The diary1Meta meta-object does check if the sender (*i.e.* pers1) has been granted access. If pers1 is not allowed to add an event to diary1, an exception is thrown. Otherwiser, the addEvent: message is performed by diary1 (step 4).

5 Related Work

5.1 AspectJ

AspectJ [16] mainly focuses on dynamic cross-cutting aspects. Nevertheless, using *inter-types declarations*, it does support to some extent the definition of static cross-

[5] The actual selector of this method is send:from:to:arguments:superFlag:-orginClass:.

cutting. Indeed, AspectJ allows the *introductions* of methods and instance variables into existing classes. However, no conflict support is provided when two aspects requires the introduction of homonymous instance variables or methods in the same classes.

Aspects reuse is also an issue with AspectJ. As demonstrated in section 2.1.2 (page 32) AspectJ does not encourage reuse. Building reusable aspects mainly relies on developers discipline and results into complex code even for simple aspects.

Last, AspectJ is complex. Dynamic and static cross-cutings are defined using different language constructs.

5.2 Hyper/J

Hyper/J stemmed from work on *Multi-Dimensional Separation of Concerns* (MDSOC) [20]. It allows developers choose arbitrary dimensions to carve up and modularize applications. Every dimension is implemented as a set of classes. Composition rules allow developers express how to merge classes defined in different dimensions.

Hyper/J shares with our work uniformly define aspects. However, our solution supports incremental dynamic weaving and unweaving. In Hyper/J weaving is a static operation that relies on program transformation. No information about original dimensions are available in the resulting application.

5.3 AspectS

AspectS is a dynamic infrastructure supporting AOP in Smalltalk [13]. It allows expressing uniformly both static and dynamic cross-cuttings. However, because AspectS' implementation relies on method wrappers, only cross-cuttings related to message dispatch can be expressed. Accesses to existing instance variables can not be captured.

Besides, new instance variables can not be introduced in application classes. Nevertheless, aspects can hold dictionaries that associate state to application objects. This solution has two limitations. Access to dictionaries is slow compared to direct access to instance variables. And, the code of aspects holding such state is rather complex.

5.4 ClassBoxes

ClassBoxes are modules allowing the definition of scopped class extensions [1]. They can be used to implement static cross-cutting [2].

This approach supports dynamic weaving/unweaving of aspects. Besides, visibility control helps resolving some potential conflicts. However, the use of ClassBoxes to implement dynamic cross-cutting is still to be studied.

5.5 Traits

Traits can be viewed as mixins without structure (no instance variables), but with a powerful composition mechanism [22]. The trait model indeed provides different operators to compose traits at methods granularity level. Developers can for example hide or rename some trait's method.

Traits can be used instead of mixins to implement unified aspects. Their composition operators can be helpful for solving conflicts among aspects. Moreover, the absence of instance variables definitions in traits reduces conflicts. However, it also restricts their expressive power.

6 Conclusion and Future Work

We described in this article foundations for a platform allowing a unified description of dynamic and static cross-cutting. To this end, we make use of mixins as aspects building blocks. In this context, weaving relies on mixin-based inheritance and reflection. Static cross-cutting is implemented using mixins that are inserted into base-level class hierarchies by the weaver. While dynamic cross-cutting is implemented using mixins are inserted into meta-level class hierarchies by the weaver. This solution helps building reusable aspects since mixins can be easily implemented without any connexion to application classes.

One possible perspective of this work is to improve conflict resolution support. Our current solution mainly relies on explicit mixins linearization. Application developers can only reorder mixins linked to a given class. Granularity of aspect conflict resolution can still be finer to allow even more conflicts resolutions. Traits compositional operators introduced by Schärli et al. [22] is a possible solution to deal with conflicts at the method level. Another alternative is to use the mixin model introduced Van Limberghen et al. [17] which provides operators to deal with both methods and instance variables conflicts.

Reuse is yet another intersting direction to follow. In this paper, we mentioned mixin reuse to build different aspects. We also, presented aspect reuse to build different applications. A third possibility yet to explore is aspect reuse to build new aspects out of existing ones.

References

[1] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. In *Research Track of the ESUG 2004 Smalltalk Conference*, Köthen (Anhalt), Germany, September 2004. Selected for publication in the special issue on Smalltalk Language of the Elsevier international journal "Computer Languages, Systems and Structures" 2005.

[2] Alexandre Bergel and Stphane Ducasse. Dynamically applying static aspects with classboxes. *Journes Francophones de la Programmation Par Aspects, JFDLPA'04*, 2004.

[3] N. Bouraqadi. Efficient support for mixin-based inheritance using metaclasses. In *Workshop on Reflectively Extensible Programming Languages and Systems at The International Conference on Generative Programming and Component Engineering (GPCE'03)*, Erfurt, Germany, September 2003.

[4] N. Bouraqadi. Metaclass composition using mixin-based inheritance. In *Research Track of the ESUG 2003 Smalltalk Conference*, Bled, Slovenia, August 2003. European Smalltalk Users Group (ESUG).

[5] N. Bouraqadi and T. Ledoux. *Aspect-Oriented Software Development*, chapter 12 – Supporting AOP using Reflection. Addison-Wesley, 2005.

[6] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311. ACM Press, 1990.

[7] Richard Cardone and Calvin Lin. Using mixin technology to improve modularity. In Filman et al. [11], pages 219–241.

[8] Olivier Caron, Bernard Carre, Alexis Muller, and Gilles Vanwormhoudt. Mise en oeuvre d'aspects fonctionnels rutilisables par adaptation. *Journe Francophone sur le Dveloppement de Logiciels Par Aspects, JFDLPA'04*, September 2004.

[9] Brian de Alwis and Georg Kiczales. Apostle: A simple incremental weaver for a dynamic aspect language. Technical Report TR-2003-16, University of British Columbia, Vancouver, Canada, March 2003.

[10] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[11] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[12] Adele Goldberg and David Robson. *Smalltalk 80*, volume 1 – The Language and its implementation. Addison-Wesley, 1983.

[13] Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[14] Sonya E. Keene. Object-oriented programming in common lisp: A programmer's guide to clos. *Addison-Wesley,Reading, Massachsetts, USA*, 1989.

[15] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Loingtier Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[16] Ramanivas Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.

[17] Marc Van Limberghen and Tom Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. *Object Oriented Systems*, 3:1–30, 1996.

[18] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, pages 147–155, Orlando, Florida, 1987. ACM.

[19] Sean McDirmid and Wilson C. Hsieh. Aspect-oriented programming with jiazzi. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 70–79, New York, NY, USA, 2003. ACM Press.

[20] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):44–50, October 2001.

[21] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. Jac: a flexible solution for aspect oriented programming in java. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of Reflection 2001*, number 2192 in LNCS, pages 1–24, Kyoto, Japan, September 2001. Springer Verlag.

[22] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003*, LNCS. Springer Verlag, July 2003.

[23] Brian C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages, POPL'84*, pages 23–35, Salt Lake City, Utah, USA, January 1984.

[24] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

Inter-Language Reflection

A Conceptual Model and Its Implementation

Kris Gybels [a] Roel Wuyts [b] Stéphane Ducasse [c] Maja D'Hondt [d]

[a] *Vrije Universiteit Brussel, Brussels, Belgium*

[b] *Université Libre de Bruxelles, Brussels, Belgium*

[c] *University of Savoie, Annecy, France*

[d] *Vrije Universiteit Brussel, Brussels, Belgium*

Abstract

Meta programming is the act of reasoning about a computational system. For example, a program in Prolog can reason about a program written in Smalltalk. Reflection is a more powerful form of meta-programming where the same language is used to reason about, and act upon, itself in a causally connected way. Thus on the one hand we have meta-programming that allows different languages or paradigms to be used, but without causal connection, while on the other hand we have reflection that offers causal connection but only for a single language. This paper combines both and presents *inter-language reflection* that allows one language to reason about and change in a causally connected way another language and vice-versa. The fundamental aspects of inter-language reflection and the language symbiosis used therein, are discussed. Moreover the implementation of two symbiotic reflective languages is discussed: Agora/Java and SOUL/Smalltalk.

Key words: Meta Programming, Reflection, Linguistic Symbiosis, Inter-Language Reflection

1 Introduction

Software engineering practices often require tools that incorporate a form of *meta programming* for extracting information from programs, checking them against a

Email addresses: `Kris.Gybels@vub.ac.be` (Kris Gybels), `Roel.Wuyts@ulb.ac.be` (Roel Wuyts), `Stephane.Ducasse@univ-savoie.fr` (Stéphane Ducasse), `Maja.D'Hondt@vub.ac.be` (Maja D'Hondt).

certain specification or generating them. Examples abound such as tools for detecting bad smells in programs to be refactored [1], checking the program's conformance with architectural restrictions [2], generating skeleton code for the implementation of design patterns or user interface elements [3], detecting the possible types a variable can hold in programs written in a dynamically typed language [4] and so on. In a meta-programming context, the *base language* is the language of the computational system under analysis, whereas the *meta language* is the language in which a representation of the base program is made available and which is used to perform the analysis. The meta language does not necessarily have to be the same as the base language. In fact, it has often been found that different programming paradigms are more suitable for certain meta-programming activities than others. Several approaches exist where the base language is for example a procedural or object-oriented one, while the meta language is based on a different paradigm: logic programming languages such as Prolog have been found to be especially suitable for extracting information from programs [1] and expressing constraints over a program's structure [5,1,6,7], the well-known tool Lint that incorporates a regular-expressions-based language for encoding patterns of typically problematic C code [8].

In the particular form of meta programming known as *reflection*, the base and meta languages are the same [9,10,11]. This contrasts with the aforementioned observation that using different programming languages as base and meta languages can be beneficial. A second crucial difference between reflection and meta programming is that in the former the base program and its representation as data in the meta program are *causally connected* [12,13]. This allows the meta program to also manipulate the elements of the base program that are made available to it, which could be the base program's source text, runtime stack, data structures in memory and so on. When the meta program only inspects the base level elements, the system is called introspective; when it also modifies them, it is called intercessory [11]. Since in either case the base and meta programs are represented in the same language, the base and meta programs can also be the same, allowing a program to manipulate itself while it is running. Reflection is heavily relied upon in the self-extensible software development systems of Smalltalk [14,15], Self [16] and CLOS [11].

This paper introduces *inter-language reflection*, a form of reflection between two different languages, possibly of a different programming paradigm. Inter-language reflection extends the causal connection property of reflection to hold between these different languages. In addition to reflection, another fundamental ingredient of inter-language reflection is *linguistic symbiosis* between the two languages. Linguistic symbiosis enables the representation of data of one language in the other, as well as the activation of behavior described in one language from the other. In order to achieve these two elements, a *data mapping* and a *protocol mapping* need to be devised between the two languages. A clean linguistic symbiosis between two languages that each have traditional reflection, results in inter-language reflection, where a program can be implemented in one language and still use the reflection

interface of the other language.

Inter-language reflection as presented in this paper is implemented in *Agora* [17] and *SOUL* [18]. Agora is a prototype-based object-oriented language that has inter-language reflection with Java, a class-based object-oriented language. This allows a very dynamic programming style in Agora, while providing access to the extensive libraries of Java. SOUL is a variant of Prolog, a logic programming language, and has inter-language reflection with Smalltalk, an object-oriented language. It has been successfully used as a tool to support several software engineering activities, such as detection of design patterns, software architectures or bad smells in source code [19], as a basis for aspect-oriented programming [20], or to reason about run-time execution traces [21]. While previous papers have demonstrated the usefulness of Agora and SOUL, this paper's purpose is to illustrate concrete instances of inter-language reflection and the specific implementation strategy. The reason why we introduce these two instances, is that Agora shows inter-language reflection of two different languages of the object-oriented programming paradigm, whereas SOUL demonstrates the more complex inter-language reflection between a language of the logic programming paradigm and a language of the object-oriented programming paradigm.

The contributions of this paper are:

- the identification of the limits of traditional reflection and meta-programming,
- the definition of inter-language reflection and its use of linguistic symbiosis,
- the presentation of a concrete implementation strategy of inter-language reflection.

The outline of the paper is as follows. Section 2 defines inter-language reflection and shows that it can be achieved through traditional reflection and linguistic symbiosis. Section 3 introduces the concept of linguistic symbiosis, and its key elements of data mapping and protocol mapping. Sections 4 and 5 show the linguistic symbiosis in Agora and SOUL, respectively. In both cases, the data mappings between the languages involved are explained. Section 6 explains how the data mapping at the base level is achieved by protocol mapping at the meta level. Section 7 introduces the concrete implementation strategy of linguistic symbiosis and inter-language reflection that is employed in both Agora and SOUL. Section 8 presents an example of inter-language reflection. Section 9 discusses related work, and Section 10 concludes the paper.

2 Inter-Language Reflection = Linguistic Symbiosis + Reflection

This paper introduces *inter-language reflection*, a form of reflection between two different languages, possibly of a different programming paradigm. Inter-language

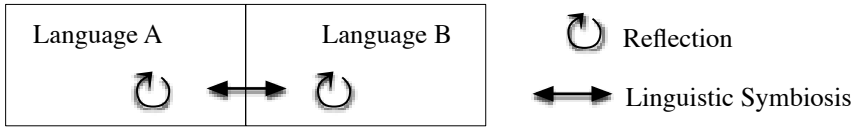

Fig. 1. Conceptual overview of inter-language reflection between two languages A and B: A and B are reflective languages that are in linguistic symbiosis.

reflection extends the causal connection property of traditional reflection to hold between these different languages. As such, a program that uses the reflection interface of the one language can be described in the other language. Therefore, inter-language reflection provides all the aforementioned benefits of being able to represent a meta program in a different programming language (or even paradigm) than the base language. Additionally a program implemented in one language, is able to make changes to elements of a program implemented in another language.

The key idea exposed in this paper is that inter-language reflection between languages *A* and *B* can be achieved by combining the following two ingredients (this is shown in Figure 1):

- *traditional reflection* of both language *A* and language *B*. Languages with traditional reflection allow programs to observe and manipulate the data of their own execution process just as if that data were regular data in the language.
- *linguistic symbiosis* between language *A* and language *B*. Two languages are in linguistic symbiosis when they can transparently exchange data and invoke each other's behavior. In order to achieve this, data of both languages needs to be mapped to one another, as well as the protocols for invoking behavior.

It is important to note that by allowing linguistic symbiosis between the two reflective languages, they can not only access each other's basic data and behavior at the base level, but also data and behavior through the reflective interfaces. Linguistic symbiosis is presented in the next section.

Note that we assume languages in which there's an explicit meta representation of the language's operations in the evaluation process. This typically holds for interpreted and bytecode interpreted languages.

3 Linguistic Symbiosis Model Overview

As illustrated in Figure 2 this section focuses on linguistic symbiosis in itself. Two languages are in linguistic symbiosis when they can transparently invoke each other's behaviour and exchange data. Linguistic symbiosis enables the representation of data of one language in the other, as well as the activation of behavior described in one language from the other. Our model for achieving linguistic symbiosis consists of two elements, a *data mapping* and a *protocol mapping* that is

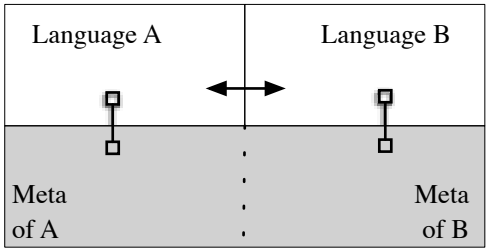

Fig. 2. Conceptual overview of linguistic symbiosis between two languages A and B, showing both base and meta levels.

needed between the two languages:

Data mapping. To achieve a tight integration at the syntactic level when passing data between the programs in the different languages, the data should "appear" in each language as seemingly native data. This means that it should be possible for programs in B to apply operations on data of A as though it was native data of B, and vice versa. Therefore operations invoked in B on data of A somehow need to be translated to operations of A, and vice-versa.

Protocol mapping. The key point in linguistic symbiosis is that the data mapping at the syntactic level comes down to a protocol mapping at the language implementation level: making the data of one language "appear" in the other is achieved on the language implementation level by ensuring that the meta representations of that data can be passed between the interpreters. To do so the protocols of the representations of both languages must be explicitly considered: to allow a meta representation to be passed to another interpreter requires making the meta operations of that interpreter applicable to that meta representation as well. Explicitly considering the protocol mapping gives a clear picture of the differences between the languages that need to be resolved in order to integrate them syntactically at the base level.

This linguistic symbiosis model does not constitute a general definition of how to construct protocol mappings for concrete languages, but serves as a conceptual framework that needs to be instantiated. The following sections therefore discuss the data mapping and protocol mapping for two concrete cases: Agora and Java, and SOUL and Smalltalk. Section 4 discusses the data mapping for Agora and Java, Section 5 discusses this for SOUL and Smalltalk, and Section 6 discusses the protocol mapping for both cases.

4 Linguistic Symbiosis between Agora and Java

Defining a linguistic symbiosis between Agora and Java requires transparent ways for exchanging data and invoking behavior. Exchanging data means that it should be possible to pass an Agora object to a Java program, and vice-versa to pass a Java

```
frame VARIABLE:
  ("java.awt.Frame" JAVA) new;

ok VARIABLE:
  ("java.awt.Button" JAVA) newString: "OK";

frame addComponent: ok;

okListener VARIABLE: [
  implements METHOD:
    (1 ARRAY: ("java.awt.event.ActionListener" JAVA));
  replaces METHOD:
    ("java.lang.Object" JAVA);
  actionPerformedActionEvent: e METHOD: {
    ("java.lang.System" JAVA) out printlnString: "Button Pressed!";
    frame setVisibleboolean: false
    }
  ];

ok addActionListenerActionListener: okListener
```

Fig. 3. Example of the language symbiosis between Agora and Java.

object to an Agora program. Invoking behavior means that from Agora it should be possible to send messages to these Java objects, and vice-versa to send messages to Java objects from within Agora programs. For these exchanges and invocations to be *transparent*, the Java object should appear as an Agora object in the Agora program: an Agora program should be able to send messages to a Java object in the same way as it sends messages to native Agora objects. The same should hold for Agora objects in Java programs.

4.1 An Example

Figure 3 shows a concrete example of an Agora program that uses linguistic symbiosis with Java:

- The first expression defines the variable `frame` to which we assign a newly created instance of the Java class `Frame`. Note that here the message `JAVA` is sent to a string that contains the name of a Java class, the message returns this Java class as an Agora object to which then the message `new` is sent.
- The second expression similarly defines a variable `ok` to hold an instance of the Java class `Button`.
- The third expression sends the message `addComponent:` to the frame with the button as argument. Note that this message is sent using the regular Agora syntax for message sending, behavior is effectively invoked on the Java object contained in the `frame` variable as if it were an Agora object.

- The fourth expression defines a variable `okListener` to hold a new Agora object to act as the button's listener.
- The fifth expression sends the message `addActionListenerActionListener:` to the button with the Agora object as argument to install this listener. Note that since the `ok` variable contains a Java object, the message and its arguments are passed to Java which in this case means an Agora object is passed to Java.

4.2 Data Mapping

Accessing Java objects from Agora. In Agora it is possible to access Java classes as *regular* Agora objects. The constructors are invoked through Agora messages to create new instances. Figure 3 shows a number of examples where a Java class is accessed from Agora using the `JAVA` message. The `JAVA` message is sent to a string, which interprets the string as the name of a class in Java and returns that class. Thus the first objects that can be "grabbed" from Agora are classes by using their name. Using Agora messages like `new` and `newString:` new instances are created. Another way for Java objects to wind up in Agora is of course by having them passed as arguments to messages to Agora objects.

Passing Agora objects to Java. Agora objects are only passed to Java when they are used as arguments in messages to Java objects from within Agora. In the last expression, the message `addActionListenerActionListener:` is sent to the Java `Button` object, with the Agora object `okListener` as argument. Because Java is a class-based language and Agora is not, the Agora object needs to appear as the instance of a Java class when it is passed to Java. Which class it is made an instance of is determined by the Agora object itself: Agora objects that are passed to Java are expected to implement the message `implements` and the message `replaces` which should respectively return an array of Java interfaces and a single Java class. From Java, the Agora object then appears as an instance of a class that implements the interfaces as given by the `implements` message and that is a subclass of the class as given by the `replaces` message. Note however that in order to preserve the dynamic nature of Agora, it is not checked whether the object actually supports the messages as declared by the Java interfaces.

5 Linguistic Symbiosis between SOUL and Smalltalk

SOUL and Smalltalk differ more fundamentally in their underlying paradigm than Java and Agora: Smalltalk is an object-oriented language while SOUL is a logic language. Achieving the linguistic symbiosis is therefore much more complicated, since the basic building blocks of each language differ: Smalltalk uses objects and messages, while SOUL uses logic terms and backtracking.

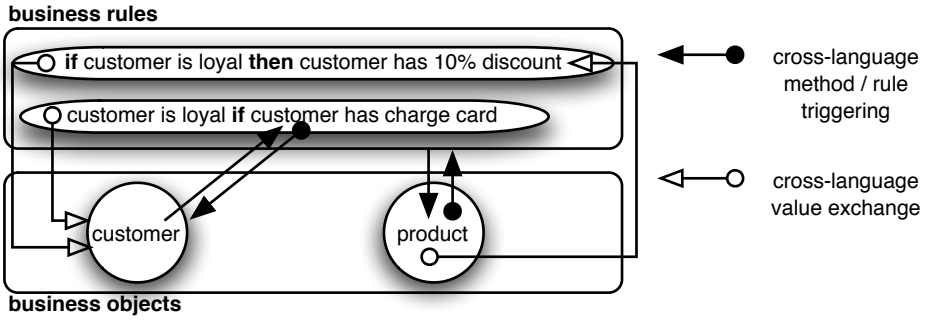

Fig. 4. Illustration of issues in defining a symbiosis between a logic and object-oriented language.

SOUL ⟷ Smalltalk

```
discount(?customer, ?product, 10) if
  loyal(?customer).
discount(?customer, ?product, 15) if
  student(?customer),
  educationalProduct(?product).

loyal(?customer) if
  [ ?customer hasChargeCard ].

educationalProduct(?product) if
  equals(?sectionitems, [ Section education products ]),
  [ ?sectionitems includes: ?product ]

...
```

```
Product>>priceFor: aCustomer
 | discounts |

 discounts :=
  (SOULEvaluator
      evaluate: 'discount(?customer, ?product, ?discount)'
      withArguments:
        (Array with: #customer -> aCustomer
                with: #product -> self))
       valuesForVariable: #discount.
 ^ price * (100 - discounts max) / 100
```

Fig. 5. Actual implementation of the business rule example in Smalltalk and SOUL

5.1 Issues and An Example

To achieve the linguistic symbiosis we again need to show how the data is mapped, and how protocol differences are resolved. These issues are illustrated in Figure 4 with an example application of symbiosis where logic rules are used to implement business rules about an object-oriented business application [22]: an object *product* calculates its discounted price for another object *customer*. A first issue to be solved is that the discount is inferred by the logic rules, which somehow have to be triggered. This is depicted by the black arrow starting from *product*. Secondly, when triggered, the logic rules need information from the objects in order to infer information. For example, one of the rules needs to invoke a method of the customer object that establishes whether *customer* has a charge card or not (denoted by the black arrows). Therefore a third issue is that the logic rules need to access the object to begin with (denoted by the white arrows). Lastly, the result of the inference of the logic rule needs to be accessible in Smalltalk again. For example, the object *product* needs to refer to the inferred *discount* of *customer*, as depicted by the white arrow starting from product.

Figure 5 shows the code for the example in SOUL and Smalltalk. In SOUL (left part of the figure), two rules are implemented for a predicate `discount`, and a third rule implements the `loyal` predicate used in the first rule for `discount`. The discount rules are triggered from the Smalltalk class `Product`'s method `priceFor:` by sending a message `evaluate:withArguments:` to the class `SOULEvaluator`. The rules get access to the *customer* and *product* objects by having them passed to the message to the `SOULEvaluator` together with an array that specifies to which logic variables the objects should be bound. The result is made accessible in Smalltalk as an object with all the possible results for the logic variable `?discount`, these solutions are accessed by sending the message `valuesForVariable:` which returns a collection with all the solutions, this collection is assigned to the variable `discounts`. The rule for the `loyal` predicate illustrates how rules can access information from the objects. This is done using a Smalltalk term, an expression that is syntactically like a Smalltalk message expression enclosed in square brackets, but it is not just a regular Smalltalk expression as it can contain logic variables to pass values from logic rules to Smalltalk methods. The only condition in the `loyal` rule is a Smalltalk term, specifying that the message `hasChargeCard` sent to the `?customer` object should return the boolean true. The `educationalProduct` rule illustrates more advanced accessing of information from the objects: in the first condition of the rule, the logic variable `?sectionitems` is bound to the result of a Smalltalk message which gets a collection of all the products of the educational section, the second condition specifies that this collection should include the product.

5.2 Data Mapping

Passing objects from Smalltalk to SOUL. Objects are generally passed to SOUL by invoking logic queries from Smalltalk and passing the objects as arguments to the query. While it is possible for logic rules to create new Smalltalk objects by sending instance creation messages to classes, this is not generally used as a logic query should normally not have any side effects.

Passing logic data from SOUL to Smalltalk. SOUL logic data can be passed to Smalltalk as arguments in Smalltalk terms, and appears in Smalltalk as objects. Message sends on these objects are simply mapped to data access operations on the logic data, thus mostly only accessor messages can be sent to these objects.

6 Linguistic Symbiosis at the Meta Level

As explained in Section 3, linguistic symbiosis involves a data mapping at the base level which is implemented as a protocol mapping at the meta level: to ensure that the interpreter of one language can apply its meta operations to the meta representations coming from the other interpreter, the protocols of these representations need

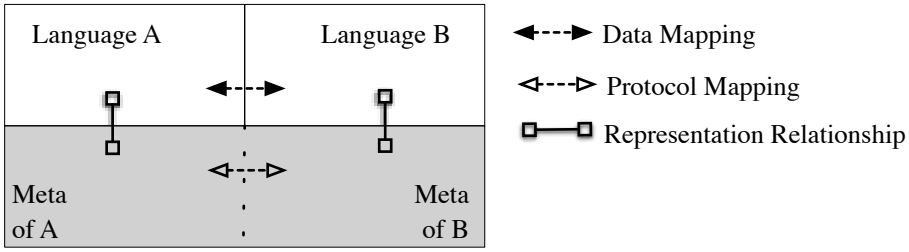

Fig. 6. Linguistic symbiosis between two languages A and B at the meta level: A and B have meta-level representations that have different protocols that need to be bridged.

to be mapped to each other. This is illustrated in Figure 6: data of languages A and B is represented at the meta level, and on this meta level the differences of protocol between the representations need to be resolved.

As before we now show how this is accomplished in Agora and Java on the one hand, and in SOUL and Smalltalk at the other hand. The choice of the meta language in which these interpreters are written doesn't really matter for demonstrating how the protocol mapping at the meta level works. We'll however also show in the next section how the conceptual model we explain here is used in actual implementations. In that case one of the two languages is actually implemented in the other and there is not a clear separation between the meta level and base level. To clearly show the difference in how the mappings occur then, we already in this section use one of the two base languages on the meta level as well: Java in the first case, Smalltalk in the second. The important point is that there is a clear separation of the base and meta levels, and that a common language is used on the meta level for the two base level languages. This is explained in further detail at the start of the next section.

6.1 Protocol Mapping for Agora and Java

On the meta level, there are two interpreters, one for Agora and one for Java. As we've assumed Java to be the meta level language for the implementation of these interpreters as well, these interpreters are written as a number of cooperating objects of different classes. Two important classes are the ones that implement the base level objects themselves: a class `JavaObject` and a class `AgoraObject`. Instances of these classes are thus meta level objects that represent base level objects.

Each of the two classes of meta objects understands a fairly similar protocol that implements the message sending of the base level. Both `JavaObject` and `AgoraObject` have methods that are the implementations of base level message sending. Of course, this protocol is similar but not entirely the same: the class `JavaObject` supports the meta operation `send(JavaMessageName, Array[JavaType],`

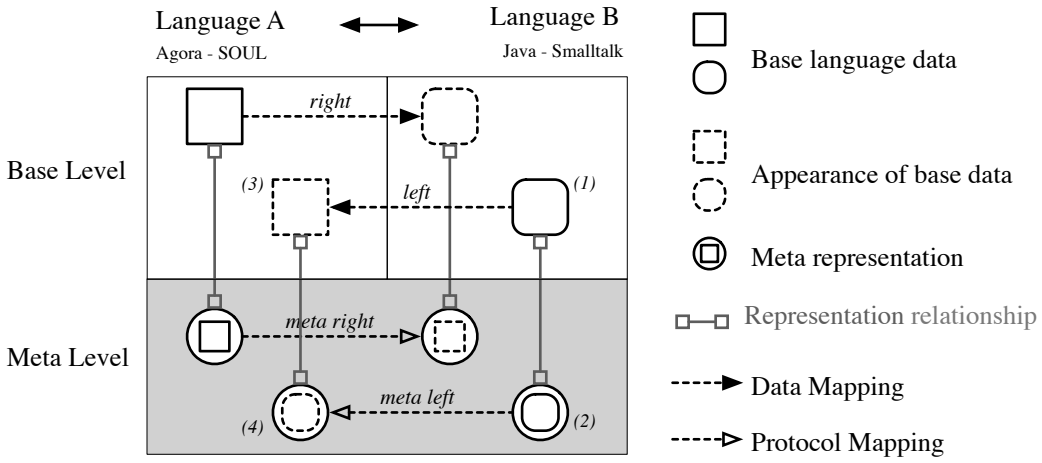

Fig. 7. Linguistic Symbiosis in more detail, focussing on the left and right appearance relationships and their equivalent relationships on the meta level.

`Array[JavaObject])` while the class `AgoraObject` supports the meta operation `send(AgoraMessageName, Array[AgoraObject])` [1].

As shown in Figure 7, the data mapping of the base level can be split in *left* and *right* appearance relationships which allow base language data of one language to appear in the other language. On the meta level, there are meta representations for this base data, and the *left* and *right* relationships of the base level require equivalent protocol mapping relationships at the meta level. A clean equivalent relationship and way of implementing the symbiosis is to introduce wrapper classes that take care of mapping the protocol differences: in the case of Agora and Java, a class `JavaWrappedAgoraObject` and a class `AgoraWrappedJavaObject` can be introduced. Instances of these classes respectively wrap around an `AgoraObject` instance and support the `JavaObject` protocol, or wrap around a `JavaObject` instance and support the `AgoraObject` protocol. So for example in the figure, the base level Java object labeled *(1)* is represented by the meta object labeled *(2)* and appears in Agora as an Agora object *(3)*, which is implemented as an `AgoraWrappedJavaObject` wrapper around the `JavaObject` instance *(2)*. One desirable property of the *left* and *right* relationships is that they cancel each other out: applying the *right* relationship to wrapped meta object produced by the *left* relationship should yield the original meta object, and vice-versa.

The protocol mappings used on the meta level between Agora and Java meta objects should have the following effect on the base level for messages between Agora and Java objects:

Sending messages from Agora to Java objects. In Agora variables, Java objects

[1] Of course, other implementations are possible as well, the ones chosen here simply illustrate the point that there is an inherent difference that requires a mapping from one to the other to enable symbiosis between the two languages.

appear as regular Agora objects, and can be sent messages in the same way as other Agora objects. For this to work a mapping is needed that maps messages sent to Java objects in Agora to Java messages, taking into account the different syntax and semantics used for messages in the two languages. The semantic difference is that the name of a message in Agora uniquely identifies a method for a specific receiver object, while in Java it does not due to the possibility for overloading. The syntactic difference is that Agora messages consist of multiple keywords after the fashion of Smalltalk, while Java messages consist of a single name. The solution adopted is to construct an Agora message from a Java message by using the type of each argument as the name for the corresponding keyword, with the exception of the first keyword which consists of the name of the Java message together with the type of the first argument. The solution for the semantic differences thus at once also provides one for the syntactic difference.

Sending `addComponent:` to `frame` is an example of an Agora message constructed from a Java message. The Java class `Frame` has at least two `add` methods: one which takes a single argument with static type `Component` and one which takes a single argument with static type `PopupMenu`. As we wish to add a "Component" in the example, the resulting mapped message that is sent from Agora to the Java object in the `frame` variable is `addComponent:`, a message consisting of a single keyword which is the concatenation of the Java message name `add:` with the name of the type of the first argument.

The same mapping is used for invoking constructors on classes, with the difference that the special name "new" is used for the constructors as they are nameless in Java. Thus to create an instance of the Java `Button` class from within Agora using the constructor that takes a `String` as an argument, the message `newString:` can be sent to the `Button` class from within Agora.

Sending messages from Java to Agora objects. When contained in Java variables, Agora objects can be sent messages to by Java objects in the same way as the latter would send messages to other Java objects. A Java message sent to an Agora object is constructed from an Agora message in the inverse way as described above.

A critical point in the mappings performed by the protocol wrappers is to ensure that the appropriate left and right relationships are applied when mapping arguments of one protocol to the other. When a `JavaWrappedAgoraObject` maps a Java `send` operation to the Agora `send` operation, the arguments involved in the message send are Java meta objects which also need to be converted to Agora ones. Thus, the mapping done by this wrapper semi-formally comes down to what is shown in Figure 8.

The rule simply describes the same protocol mapping solution for sending Java messages to Agora objects as described above, but illustrates the point of needing to convert the receiver and arguments to Agora objects. Applying the left relationship on the receiver, which in this case is the `JavaWrappedAgoraObject` wrapper, simply results in the unwrapped Agora meta object. Similarly, the left relationship

receiver.`send(`"*name*", {*type1*, *type2*, … *typen*},
{*argument1*, *argument2*, … *argumentn*}`)`
==
right[*result*]

⇳

left[*receiver*].`send`("*nameType1Name:type2Name:…typenName*",
{left[*argument1*], left[*argument2*], …left[*argumentn*]})
==
result

Fig. 8. Semi-formal description of meta operation mapping for the send operation from Java to Agora.

applied to the arguments either wraps them or unwraps them, depending on whether they were wrapped Agora meta objects produced by the right relationship, or plain Java meta objects in the first place. As also illustrated, the result of the mapped message also needs to be mapped back using the *right* relationship to turn it from an Agora object into a Java object.

The converse rule for mapping Agora messages to Java messages is very similar and can be given without further explanation as illustrated in Figure 9, note that this rule is easily derived from the rule above using the fact that the *left* and *right* relationships cancel each other out (*i.e.* $left[right[x]] = x$).

receiver.`send(`"*nameType1Name:type2Name:...typenName*",
{*argument1*, *argument2*, … *argumentn*})
==
left[*result*]

⇳

right[*receiver*].`send(`"*name*", {type1, type2, …, typen},
{right[*argument1*], right[*argument2*], … right[*argumentn*]}`)`
==
result

Fig. 9. Semi-formal description of meta operation mapping for the send operation from Agora to Java.

6.2 Protocol Mapping for SOUL and Smalltalk

Unifying objects. In an interpreter for a logic language like SOUL, unification is a particularly important operation that is applied on the meta representations for logic rules, logic functors and other logic terms. In the actual implementation of SOUL, there is a class `AbstractTerm` from which all classes for representing the different kinds of logic terms - functors, variables and lists - inherit. The `AbstractTerm` class defines an abstract method `unifyWith:inEnvironment:`

which the other classes implement accordingly. The method is passed another object that is the meta representation of another logic functor or list etc. and an environment of logic variable bindings. The method on meta objects representing logic functors for example checks if the other object also represents a logic functor, and then recursively sends the same unification messages to the different objects representing the arguments of each of the two functors. A meta object representing a logic variable responds to the message by checking whether the environment already holds a binding for it. If not it simply adds a new binding with the meta object it is being unified with as the binding's value. If there already is a binding, the logic variable object again recursively sends the unification meta message to the existing binding's value with the same arguments it received for the message itself, which are the environment and the logic meta object it should unify with. Thus unification is implemented by a protocol of recursive message sending, and the meta objects passed as arguments to the unification message are expected to support this protocol.

An interpreter for Smalltalk on the other hand has meta representations for objects and classes. The meta objects representing Smalltalk objects need to support a protocol for sending messages, accessing instance variables etc. The meta objects representing classes need to support a protocol for looking up methods, defining instance variables etc. Thus, in an interpreter for Smalltalk, there would be a class `AnObject` whose instances would represent objects at the base level. The `AnObject` meta objects would understand messages such as `sendSelector:withArguments:`.

Thus, the base-level *left* appearance relationship which allows a base-level Smalltalk object to appear in SOUL, needs an equivalent on the meta level which maps the unification protocol to the message send protocol. The particular solution chosen in SOUL for this issue is to allow objects to unify when they are equivalent according to the equivalency message `=`. One way to implement the left relationship on the meta level then, is to put a wrapper around Smalltalk meta objects with the wrapper mapping the unification operation to the message send operation. In SOUL, this wrapper is `SmalltalkObject`, a subclass of `AbstractTerm`.

As with similar mappings in Java and Agora, care must again be taken to perform the appropriate *right* relationship on any meta objects involved in the mapping. Thus, when a `SmalltalkObject` wrapper receives a `unify:` message, it maps this operation to the message send operation `send:withArguments:` where the message that is sent is `=`. Both the receiver of this operation and the argument that it is passed need to support the message send protocol of Smalltalk meta objects, thus the *right* relation needs to be applied before applying the message send operation. Semi-formally, the protocol difference mapping that happens on the meta level is:

lo1 `unify:` *lo2*

⇳

right[*lo1*] `send: #= withArguments:` { right[*lo2*] }

Sending messages from SOUL to Smalltalk objects. Message sending is again not an operation native to logic programming, but depending on the type of the message it can be mapped to either of two operations of logic programming. Boolean messages can be naturally mapped to proving of conditions in logic rules, while accessing data from objects by means of accessor or other side-effect-less methods can be mapped to a part of unification. Invoking mutator and other methods with side-effects does not make sense from the logic paradigm.

The mapping of the first two message types is handled in SOUL with the same linguistic construct: the Smalltalk term. A Smalltalk term is a novel linguistic construct that was added to SOUL specifically for symbiosis: as shown in the rule for the predicate `loyal` in the example, a Smalltalk term is denoted by square brackets ("`[`" and "`]`") and contains a message send in Smalltalk syntax but involving logic variables. A smalltalk term can be used as a condition in a rule, as also shown in the rule for the `loyal` predicate, in which case the condition proving operation of logic programming maps it to the evaluation of the message of the Smalltalk term; the message is expected to return a boolean which is then mapped to the success or failure of proving the condition. A Smalltalk term can also be used as an argument in a condition, or as part of other compound data structures of logic programming such as lists, in which case the unification operation maps it to execution of the message: when a Smalltalk term is unified with another logic data construct, this is first mapped to evaluating the message in Smalltalk and the resulting object is then again unified with the other logic data construct using the process described in the previous point.

Invoking SOUL logic queries from Smalltalk. The invocation of logic queries is mapped to a message send to a `SOULEvaluator` class in Smalltalk. The `SOULEvaluator` class supports a message `evaluate:withArguments:` which can be passed a query as a string. The second argument of this message is an array which is used to specify which objects are passed to the query for which logic variable: the array should contain associations of names of logic variables to objects. The message finds all solutions for the given query in SOUL.

The result of the `evaluate:withArguments:` message requires another mapping because of a particular difference between SOUL and Smalltalk: logic queries can have several "output" variables and furthermore can result in multiple different results for these variables, while Smalltalk messages can only return a single object. The results of the query are therefore mapped to a single object that understands a message `valuesForVariable:` which expects as argument the name of a logic variable used in the query. The result of this message is the solutions of the query for that variable, mapped to a collection object.

Accessing SOUL data from Smalltalk. In logic programming, data is accessed from compound structures such as functors and lists through unification, this needs to be mapped to Smalltalk's accessor messages for accessing data. When a SOUL value is passed to Smalltalk, it appears in Smalltalk as an instance of the equivalent Smalltalk class for that type of SOUL value: lists appear as instances of `OrderedCollection`, numbers as `Number` instances etc. Functors are mapped to instances of the SOUL-specific class `CompoundTerm`, which sup-

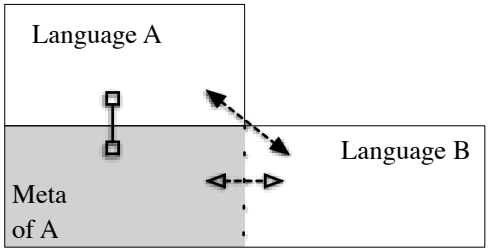

Fig. 10. Implementation of inter-language reflection and language symbiosis.

ports messages for accessing the functor's name and its arguments.

7 Inter-Language Reflection in Actual Implementations

Our conceptual model for inter-language reflection and linguistic symbiosis is readily applicable to actual implementation schemes where the two languages in symbiosis are implemented as interpeters in a third common implementation language. There are however two differing schemes possible, and in this section we explain how the conceptual model maps to these schemes. The first possible variation is that the interpreters are not written in a common implementation language. The second variation is that the interpreter of one language is written in the other language, and that a linguistic symbiosis is defined between the first language and its implementation language rather than with a language that is also implemented in that implementation language.

As noted earlier, the first variation simply shifts the problem of achieving a linguistic symbiosis one level down. A key point in our conceptual model is to explicitly take into account the meta level for both of the two base level languages, and to assume that at the meta level there is a common implementation language. This allows us to clearly show how data mapping at the base level comes down to protocol mapping at the meta level: while the meta representations of each language can already be exchanged between the two interpreters because of the common language, they support a different protocol of meta representations which needs to be mapped. In the actual implementation variation where the meta representations of one language are written in language X and those of the other in language Y, there needs to be a linguistic symbiosis between X and Y to allow the meta representations to be interchanged in the first place.

The second variation essentially entails that the meta level of one language is made to overlap the base level of the other language. As illustrated in Figure 10, the interpreter for the one language is written in the other language, and there is no interpreter for the other language at this level. One reason for this variation is that in practice it is typically easier to implement a new language in the one with which it should be in an inter-language reflection relationship, rather than in a common

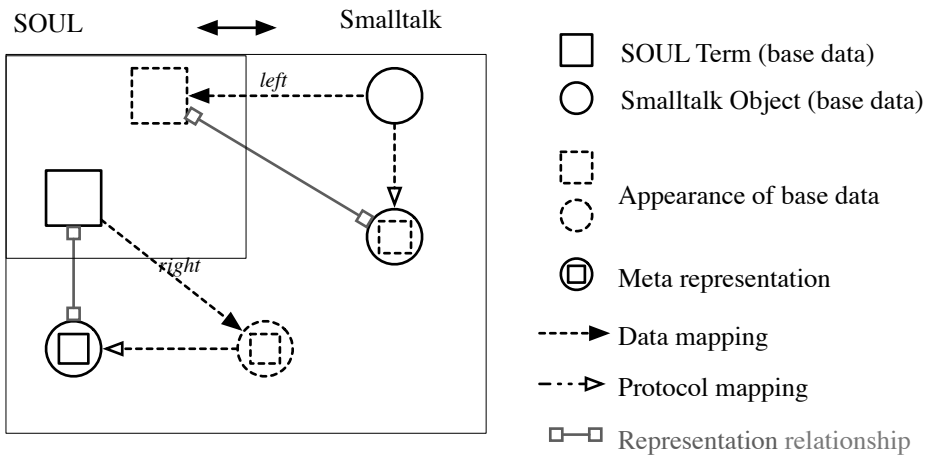

Fig. 11. Folding of inter-language reflection and language symbiosis in the actual implementation of SOUL in Smalltalk.

language, or that such an implementation already exists and there is a need to allow for inter-language reflection. Note that while we already used one of the base languages as meta language as well in the explanation of the conceptual model in Section 3, we still made a distinction between the meta level and the base level. The variation we are referring to here is that, as illustrated in Figure 11, the meta representations of one language – SOUL in the figure – exist on the same level as the values of the other language. This deviation of the conceptual model has an effect on how the linguistic symbiosis and inter-language reflection are actually implemented, as we'll discuss in more detail in the remainder of this section.

7.1 Linguistic Symbiosis Implementation

One effect of the base and meta level overlap is that the *right* relationship maps a SOUL (or Agora) value *directly* to a wrapper. Contrast this with the pure conceptual model of Figure 7, where the *right* relationship allows a SOUL value to appear in Smalltalk, and this appearance is implemented as a wrapper around the meta object representing the value. Here, the *right* relationship maps directly to the wrapper. Furthermore, this wrapper is a base level Smalltalk object, rather than a meta level object as in the pure conceptual model. Thus the wrapper translates *base* level Smalltalk (or Java) messages to *meta* operations on the SOUL (or Agora) meta object.

The important point to note about this difference in how the mappings work is that the mappings in the conceptual model more clearly show the protocol difference that is being solved. This is the reason for clearly separating the base level and meta level in the conceptual model. For example, the mappings in the conceptual model show that the Java message send operation takes the static type of the arguments into account, while the one for Agora does not. Because in the actual

implementation, wrappers map between base level and meta level operations, this difference is no longer as obvious. The mapping performed by the *right* wrappers in the actual implementation for Agora and Java is for example the one given in figure 12. Note again that the *base* level Java message with name `name` is mapped to the Agora meta operation `send`, while previously it was the Java *meta* operation `send` that was mapped to the Agora meta operation `send`.

receiver.`name`(*argument1, argument2, … argumentn*)
==
right[*result*]
⇳
left[*receiver*].`send`("*nameType1Name:type2Name:…typenName*",
{left[*argument1*], left[*argument2*], …left[*argumentn*]})
==
result

(where {type1 ... typen} are the static types of the arguments)

Fig. 12. Semi-formal description of base and meta level operation mapping in actual implementations with overlap of base and meta levels.

The *left* relationship is similarly affected. Making a Smalltalk object appear in SOUL for example involves wrapping the Smalltalk object in a wrapper that maps the SOUL meta protocol to the Smalltalk base level, instead of as in the conceptual model where it maps it to the Smalltalk meta protocol.

7.2 Inter-Language Reflection Implementation

The effect of the overlap in the actual implementations is also of folding the inter-language reflection of one direction. To reflect about one language from within the language that implements it does not require getting a reflective representation in the one language and passing it through symbiosis to the other. So in the case of SOUL in Smalltalk, there is no need for the (base level) Smalltalk to access a reflective representation of SOUL by having SOUL reflect on itself and passing the reflective representation by linguistic symbiosis to Smalltalk. Instead, for Smalltalk objects to reflect on SOUL only requires them to directly access the Smalltalk objects representing SOUL *without* applying the *right* relationship. This can actually be another reason for deviating from the conceptual model in the actual implementation. As it is thus for one language not necessary to be reflective. Note however, that in the other direction where SOUL is used for inter-language reflection about Smalltalk the combination of linguistic symbiosis and traditional reflection is still used to achieve this inter-language reflection.

8 Examples

A very simple example of using inter-language reflection in SOUL about Smalltalk is one for generating accessor methods on a class. While simple, the example nevertheless clearly shows the use of linguistic symbiosis for inter-language reflection. More extensive examples can be found in previous publications on the use of SOUL [1,2,6,21,23]. The following rule defines what the source for an accessor method for a variable of a class should be:

```
accessorMethod(?class, ?varname, ?source) if
   hasInstanceVariable(?class, ?varname),
   equals(?source, [ ?varname , ' ^ ' , ?varname ])
```

The first condition of the above rule specifies that `?varname` should be the name of an instance variable of the class `?class`. The predicate `hasInstanceVariable` is one from a library that comes with SOUL with several such predicates, the rules for this predicate are not shown here, but we can note they make use of linguistic symbiosis to retrieve the instance variables from the class. The second condition of the above rule specifies that the `?source` variable should contain a string which is the concatenation of the variable name, with a return expression that returns the variable's value. Note that this concatenation is defined using an argument to the `equals` predicate where linguistic symbiosis is used: the value in `?varname` is *right* mapped to Smalltalk, which in this case will result in a string, to which then the concatenation message "`,`" is sent, the resulting string is *left* mapped back to SOUL.

This rule can then be used in a query to actually generate all the accessor methods for a class, for example by finding all solutions to the query below one can generate all accessor methods on the `BankAccount` class:

```
if accessorMethod([BankAccount], ?variable, ?body),
   [ ?class compile: ?body. true ]
```

The above query shows two things. First of all, it clearly shows the use of the combination of traditional reflection and linguistic symbiosis for doing inter-language reflection: the `compile:` message that is sent is a reflective message in Smalltalk and it is sent from SOUL using linguistic symbiosis.

Secondly, the query shows that both introspection and intercession using inter-language reflection about Smalltalk is possible from SOUL. While doing side-effects breaks the declarative nature of logic programming and is therefore usually not recommended, there is no inherent restriction in SOUL that prevents one from accessing Smalltalk's intercessory reflection interface. Thus it is possible to send such messages as `compile:` and other messages that change the Smalltalk program, as well as all the messages for simply querying the Smalltalk program for

lists of instance variables of a class, or its subclasses etc.

9 Related Work

The term "linguistic symbiosis" was previously defined in the work on RBCL [24]. RBCL is a language implemented in C++ which also has linguistic symbiosis with C++. This is used to allow RBCL base level objects to interact with, and take the place of the RBCL meta objects defined in C++, thus achieving what we have dubbed traditional reflection. There are three important differences with our work. Firstly, the use of linguistic symbiosis as a mechanism for achieving inter-language reflection was not considered. Secondly, the languages between which symbiosis was defined were not fundamentally paradigmatically different as in our case for SOUL and Smalltalk. Lastly, in RBCL the symbiosis was directly defined in the implementation with overlap of the RBCL meta level and C++ base level, there was no consideration of the conceptual model we introduced where the linguistic symbiosis is defined through a common meta level for the two languages which allows a clearer modeling of which meta operations need to be mapped and how.

While Agora was one of the first languages to be implemented and integrated in Java, the popularity of the platform has brought about numerous languages which are hosted in Java. In most cases, linguistic symbiosis and our model for it was not explicitly considered. In several cases, the integration is asymmetric: from the hosted language, Java classes can be instantiated and the instances can be sent messages, but only Java objects can be passed as arguments, not objects implemented in the hosted language (*i.e.*, the integration is parasitic instead of symbiotic). In other cases, the integration is not fully asymmetric, but the use of values from the hosted language in Java is not transparent. For logic languages integrated in Java, an extensive survey illustrating these problems is given by D'Hondt [25].

The integration of Piccola in Java (JPiccola) and also in Smalltalk (SPiccola) is very interesting in this respect, as it is an exceptional case in which a concept similar to linguistic symbiosis, "inter-language bridging", was considered for achieving the integration [26]. However, while in the definition of "inter-language bridging" the meta-level of both languages is also considered, the meta and base levels are not clearly separated which has resulted in an asymmetric integration. The meta operations applied by the interpreters are not explicitly considered: the definitions of the *up* and *down* operations, which are respectively equivalent to our *left* and *right* operations, is only given in terms of base level data mapping rather than as protocol mapping at the meta level. The *up* operation maps a Smalltalk object to a Piccola "form" with a "service" - the behavior of the form - for each method of the object. The *down* operation is described as a simple mapping in which "the form itself is passed down to the host language". This actually has the semantics of mapping a Piccola form to its meta representation in Smalltalk, the services of the form are

thus not mapped to Smalltalk methods and a Piccola form cannot be sent messages from Smalltalk to invoke the services as if the form is a regular Smalltalk object. The *down* operation thus confuses the base and meta levels and actually is a folding of the linguistic symbiosis and inter-language reflection operations as we've described in Section 7. The integration is thus asymmetric as it is not possible to implement a form in Piccola which when passed "down" behaves as an existing Smalltalk object, while in the other direction it is possible to implement a Smalltalk object which when "upped" behaves the same as a native Piccola form. An interesting aspect of Piccola is that it allows reflective control of the *up* operation: instead of simply returning the "upped" object, the *up* operation can be modified to return a Piccola form which has a field "peer" with the actual "upped" object. This can be used to give the "upped" object a Piccola-specific interface by implementing services that appropriately forward to the peer. However, the *up* and *down* operations behave differently on such forms and do not have the desirable property of cancelling each other out (cfr. Section 6.1): applying *down* on such a form results in the "downed" peer, when that is again "upped" it is not the original "downed" form. This resulted in the need for a reflective control over the *down* operation as well, so forms can be explicitly "protected" in certain cases when they are downed [26].

A specific goal of the .NET platform is "language inter-operability". For this, a common intermediate language was defined, to which all languages supported on the platform are compiled. The common intermediate language is actually a more primitive form of the major language of the platform, C#. Compiling a language to .NET is thus in part a base level data mapping to integrate with C#, which does not clearly expose the meta level protocol differences of C# and the implemented language.

One can hardly talk about reflection without discussing the work that has been done in the LISP community, even though the goals of our approach are quite different. One of the very nice features of LISP is that it has a built-in mechanism to represent its language constructs: the quotation form. This provides a core meta-reasoning structure, since parts of programs can be assembled, passed around and then evaluated at will. This basic LISP functionality was extended in the well-known work on *procedural reflection* by Smith [27]. In this work, 2 languages were introduced. The first, 2-LISP, deals with quotation issues by providing two explicit user primitives to switch between representations of structures and the structures themselves. These primitives were called *up* and *down*, the equivalents of the *left* and *right* relationships used in this paper. We want to stress two problems with 2-LISP, which is that *up* and *down* needed to be called by the user whenever necessary and that *down* is not the inverse of *up*. 2-LISP was actually meant as the basis for the better-known 3-LISP, a reflective language with an implementation based on *reflective towers*.

Muller has argued that the quotation form in the original definition of LISP is essentially flawed [28], and hence that the apply in LISP and descendants like 2-

LISP and 3-LISP or even Scheme *crosses levels*. Moreover, it is this level-crossing that allows much of the meta-circular capabilities of LISP. This was remarked by Muller, and was addressed in his LISP flavor, called M-LISP. In M-LISP, the apply function does not cross levels, which removes a lot of the awkward constructions needed in 2-LISP. Moreover, up is represented by a relation R, and down by R^{-1}, where down is the inverse of up. Reification has to be introduced at the cost of equational reasoning (which is done with extended M-LISP), and, even extended M-Lisp corresponds to only a restricted 3-Lisp.

The approach taken by inter-language reflection is comparable with the approach taken by M-Lisp, except that the goals are quite different. The goal of M-Lisp (as with the other research in reflection in general) is to study 'self-extensibility' of programs, and provide formal language semantics to that end. The approach in our paper is driven from a software engineering problem, to study a symbiosis between two languages from different paradigms. In our case, *left* and *right* therefore not only bridge levels, but also language paradigm boundaries. As in M-Lisp (and contrary to 2-Lisp and 3-Lisp), *left* and *right* in SOUL are symmetric and are called automatically. Note however that in SOUL this relation is still kept fairly simple (only objects are reified as logic terms), however this is not necessarily the case. One of the important parts of future work is to have a tighter integration between languages by reifying more object-oriented concepts, which leads to a more difficult relation.

10 Conclusion

In this paper we introduced inter-language reflection, an extension of traditional single-language reflection to two languages. We introduced a scheme for achieving inter-language reflection where traditional reflection is combined with linguistic symbiosis. Linguistic symbiosis allows programs in two languages to transparently interchange values and invoke behavior defined on these values in the other program. As traditional reflection already allows programs to invoke behavior and access values that are causally connected to its own execution process, the combination with linguistic symbiosis automatically allows programs in each language to access the traditional reflective interface of the other language.

We introduced a conceptual model for linguistic symbiosis where, in contrast with previous work on linguistic symbiosis, we made the meta level of both base level languages explicit. This model thus allows us to clearly show how achieving linguistic symbiosis on the base level comes down to solving protocol differences between the meta operations applied on the meta level. On the meta level, different meta operations are applied to the meta representations for each language, and allowing the values on the base level to be interchanged requires making the meta operations for one language applicable to the meta representations of the other. We

illustrated this for two cases of linguistic symbiosis, that of SOUL with Smalltalk, and of Agora with Java.

We showed how in actual implementations of inter-language reflection and linguistic symbiosis, the clear separation made in the conceptual model between base level and meta level is typically abandoned. This simplifies the actual implementation of inter-language reflection as the base level of one of the two languages can directly access the meta representations of the other. We showed how this affects actual implementations of linguistic symbiosis to involve wrappers that map base and meta operations rather than meta operations as in the conceptual model. But this folding of one meta level makes the linguistic symbiosis mechanism harder to understand. The simpler conceptual model helps to see the mechanisms in their pure form and subsequently to understand the 'shortcut' taken in typical implementations.

References

[1] Wuyts, R.: Declarative reasoning about the structure object-oriented systems. In: Proceedings of the TOOLS USA '98 Conference, IEEE Computer Society Press (1998) 112–124

[2] Mens, K., Wuyts, R., D'Hondt, T.: Declaratively codifying software architectures using virtual software classifications. In: Proceedings of TOOLS-Europe 99. (1999) 33–45

[3] Florijn, G., Meijers, M., van Winsen, P.: Tool support for object-oriented patterns. In Aksit, M., Matsuoka, S., eds.: Proceedings ECOOP '97. Volume 1241 of LNCS., Jyvaskyla, Finland, Springer-Verlag (1997) 472–495

[4] Spoon, S.A., Shivers, O.: Demand-driven type inference with subgoal pruning: Trading precision for scalability. In: Proceedings of ECOOP'04. (2004) 51–74

[5] Minsky, N.H.: Law-governed regularities in software systems. (1994)

[6] Wuyts, R., Mens, K.: Declaratively codifying software architectures using virtual software classifications. In: Proceedings of TOOLS-Europe 1999. (1999)

[7] Crew, R.F.: Astlog: A language for examining abstract syntax trees. In: Proceedings of the USENIX Conference on Domain-Specific Languages. (1997)

[8] Johnson, S.C.: Lint, a C program checker. Computing Science TR **65** (1977)

[9] Ferber, J.: Conceptual reflection and actor languages. In North-Holland, P.M., Nardi, D., eds.: Meta-level Architectures and Reflection. (1988) 177–193

[10] Bobrow, D., Gabriel, R., White, J.: Clos in context — the shape of the design. In Paepcke, A., ed.: Object-Oriented Programming: the CLOS perspective. MIT Press (1993) 29–61

[11] Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press (1991)

[12] Maes, P.: Concepts and experiments in computational reflection. In: Proceedings OOPSLA '87, ACM SIGPLAN Notices. Volume 22. (1987) 147–155

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

[14] Rivard, F.: Reflective Facilities in Smalltalk. Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février 1996 (1996)

[15] Ducasse, S.: Evaluating message passing control techniques in Smalltalk. Journal of Object-Oriented Programming (JOOP) **12** (1999) 39–44

[16] Ungar, D., Smith, R.B.: Self: The power of simplicity. In: Proceedings OOPSLA '87, ACM SIGPLAN Notices. Volume 22. (1987) 227–242

[17] Meuter, W.D.: Agora: The story of the simplest mop in the world. In: Prototype-based Programming, Springer-Verlag (1998)

[18] Wuyts, R.: A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation. PhD thesis, Vrije Universiteit Brussel (2001)

[19] Wuyts, R., Ducasse, S.: Symbiotic reflection between an object-oriented and a logic programming language. In: ECOOP 2001 International Workshop on MultiParadigm Programming with Object-Oriented Languages. (2001)

[20] Gybels, K.: Aspect-Oriented Programming using a Logic Meta Programming language to express cross-cutting through a dynamic joinpoint structure. Licentiate's thesis, Vrije Universiteit Brussel (2001)

[21] Roover, C.D., Gybels, K., D'Hondt, T.: Towards abstract interpretation for recovering design information. Electronic Notes in Theoretical Computer Science **131** (2005) 15–25

[22] D'Hondt, M., Gybels, K.: Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In: Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC 2004), Special Track on Object-Oriented Programming, Languages and Systems, ACM Press (2004)

[23] Fabry, J., Mens, T.: Language-independent detection of object-oriented design patterns. In: Elsevier International Journal: Computer Languages, Systems and Structures. (2003)

[24] Ichisugi, Y., Matsuoka, S., Yonezawa, A.: Rbcl: a reflective object-oriented concurrent language without a runtime kernel. In: IMSA'92 International Workshop on Reflection and Meta-Level Architectures. (1992)

[25] D'Hondt, M.: Hybrid Aspects for Integrating Rule-Based Knowledge and Object-Oriented Functionality. PhD thesis, Vrije Universiteit Brussel (2004)

[26] Schärli, N.: Supporting pure composition by inter-language bridging on the meta-level. Master's thesis, Philosophisch-naturwissenschaftlichen Fakultät der Universität Bern (2001)

[27] Smith, B.: Reflection and semantics in lisp. In: Proceedings of the 11th Symposium on Principles of Programming Languages. (1984) 23–35

[28] Muller, R.: M-LISP: A representation-independent dialect of LISP with reduction semantics. ACM Transactions on Programming Languages and Systems **14** (1992) 589–616

Runtime Bytecode Transformation for Smalltalk ⋆

Marcus Denker [a] Stéphane Ducasse [b] Éric Tanter [c]

[a] *Software Composition Group*
IAM — Universität Bern, Switzerland

[b] *Software Composition Group*
IAM — Universität Bern, Switzerland, and
Language and Software Evolution Group
LISTIC — Université de Savoie, France

[c] *Center for Web Research, DCC*
University of Chile, Santiago, Chile

Abstract

Transforming programs to alter their semantics is of wide interest, for purposes as diverse as off-the-shelf component adaptation, optimization, trace generation, and experimentation with new language features. The current wave of interest in advanced technologies for better separation of concerns, such as aspect-oriented programming, is a solid testimony of this fact. Strangely enough, almost all proposals are formulated in the context of Java, in which tool providers encounter severe restrictions due to the rigidity of the environment. This paper present BYTESURGEON, a library to transform binary code in Smalltalk. BYTESURGEON takes full advantage of the flexibility of the Squeak environment to enable bytecode transformation at runtime, thereby allowing dynamic, on-the-fly modification of applications. BYTESURGEON operates on bytecode in order to cope with situations where the source code is not available, while providing appropriate high-level abstractions so that users do not need to program at the bytecode level. We illustrate the use of BYTESURGEON via the implementation of method wrappers and a simple MOP, and report on its efficiency.

Key words: Smalltalk, object-oriented programming, bytecode transformation, metaprogramming

⋆ We acknowledge the financial support of the Swiss National Science Foundation for the project "A Unified Approach to Composition and Extensibility" (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006) and "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006). É. Tanter is financed by the Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

Email addresses: `denker@iam.unibe.ch` (Marcus Denker),

1 Introduction

Many objectives of software engineering can be served by appropriate program transformation techniques. Software adaptation can be used for Binary Component Adaptation (BCA), a technique proposed by Keller and Hölzle which relies on coarse-grained alterations of component binaries to make them interoperable [1]. Another objective of software adaptation is that of separation of concerns [2], as first emphasized by work carried out in the reflection community [3–5], and more recently, aspect-oriented programming (AOP) [6]. In this context, transformation techniques are used to merge together different pieces of software encapsulating different concerns of the global system. Program transformation is a valid implementation techniques for reflection and AOP when an open interpreter of the considered language is not available.

Fine-grained control of computation, such as message passing control in the context of object-oriented programming, is the corner stone of many interesting applications [7]. It has been used for a wide range of application analysis approaches, such as tracing [8–10], automatic construction of interaction diagrams, class affinity graphs, test coverage, as well as new debugging approaches [11, 12]. Message passing control has also been used to introduce new language features in several languages, for instance multiple inheritance [13], distribution [14–16], instance-based programming [17], active objects [18], concurrent objects [19], futures [20] and atomic messages [21, 22], as well as backtracking facilities [23].

CLOS is one the few languages that offers a dedicated metaobject protocol supporting language semantics customization [24]. Other languages such as Smalltalk and Java rely on techniques or libraries to either transform code or take control of the program execution [7, 25]. The most basic way to alter programs is of course to modify the source code and recompile it. This approach is used by several Java systems, such as OpenJava [26] and the Java Syntactic Extender [27]. However, in many contexts, relying on the availability of source code is limiting since most applications ship in binary form, and in open distributed systems, source code is usually not known in advance. Furthermore, the source language from which the actual binary was obtained is not necessarily the mainstream language of the runtime system. Bytecode manipulation, as done in the Java world by tools such as BCEL [28] and Javassist [29] is a particularly pertinent alternative. The challenge is to provide appropriate high-level abstractions to bytecode transformation, in order to shield users from the burden of working at the bytecode level [25].

To the best of our knowledge, there is no single bytecode transformation tool for the Smalltalk/Squeak environment, in the line of what Javassist represents for the

`ducasse@iam.unibe.ch` (Stéphane Ducasse), `etanter@dcc.uchile.cl` (Éric Tanter).

Java world. This is all the more surprising that the Squeak environment actually represents an ideal environment for bytecode transformation. In contrast with Java where full bytecode transformation is only possible at load time, and very severely limited at runtime, Squeak enables the full power of bytecode transformation to be used dynamically. The purpose of BYTESURGEON is precisely to leverage the flexibility of the Smalltalk language and the Squeak environment to provide a backend to designers of toolkits for component adaptation, reflective and metaprogramming, and aspect-oriented programming.

The contributions of this paper are:

- a motivation for the need of a dynamic bytecode transformation framework for Smalltalk, working at appropriate levels of abstraction,
- a framework, called BYTESURGEON, that enables *runtime* bytecode transformation via a two level API,
- a simple MOP that can be used to compare bytecode transformation frameworks.

The paper is organized as follows: Section 2 explains the need for bytecode manipulation at appropriate levels of abstraction, by discussing related work. Then we present BYTESURGEON at work in Section 3. Section 4 details some aspects of the architecture. In Section 5, we validate the interest of our framework via the implementation of two language features: method wrappers [10], and a simple runtime metaobject protocol (MOP) making use of runtime manipulation for dynamic (un)installation of hooks; a first set of benchmarks completes the validation of BYTESURGEON. Section 6 discusses future work and concludes.

2 The Need for Bytecode Manipulation

There are many ways to change the semantics of programs, ranging from code preprocessing to modification of the language runtime environment. If the language runtime is not an open implementation offering an adequate metaobject protocol (MOP) [24], then modifying it directly sacrifices portability; since mainstream Smalltalk virtual machines such as Squeak are not open in this sense, we discard the alternative of intervening at the VM level.

Source code transformation can be done either directly on the text (concrete syntax) or on the abstract syntax tree (abstract syntax). Furthermore, in language environments where source code is compiled to an intermediate bytecode language which is abstract enough, bytecode transformation is an interesting approach; it is actually widely used in the Java community.

In Section 2.1 we discuss the inconveniences of source code approaches. Still, once bytecode transformation is agreed upon, the issue of the abstraction level offered to

the programmer appears, discussed in Section 2.2. We also discuss the limitations of bytecode transformation in the context of Java.

2.1 Disadvantages of Source Code Transformation

Transforming source code at the concrete syntax level is typically avoided because of the lack of structure and abstraction at the text level. Transformation of abstract syntax trees (ASTs) is much more adequate, but still suffers from a number of limitations.

No access to the source code. For the sake of saving space or ensuring a first level of privacy, the source code of an application is usually not distributed. Using source code strippers or removing symbolic information are current practices to reduce the size of an application before deployment. Furthermore, in open contexts such as mobile agent platforms and open distributed systems, code is typically not known in advance. One can of course rebuild an AST from bytecode, but this technique presents a number of challenges: bytecode-to-AST decompiling is a slow process, and typically requires the decompiler to know about bytecode generation patterns used by the compiler so as to rebuild meaningful AST nodes.

No original language warranty. Most mainstream languages today, such as Java, Squeak and C#, are based on a virtual machine executing bytecodes, and these virtual machines are actually used as the execution engines of various languages, other than the "original" ones. For instance, for the Croquet environment [30], a number of experimental scripting languages have been developed, among them languages similar to JavaScript and LOGO. Another example is the Python language, which can be compiled to Java bytecodes [31]. To provide practical performance, these languages come with their own custom compiler that produces bytecode for a production-quality virtual machine. Therefore a code transformation tool working at the AST level rebuilding the AST from bytecode would require a custom decompiler. On the other hand, working on bytecode, although lower-level than AST, makes it possible to uniformly apply transformations even in the presence of non-original languages.

Recompiling is slow. Finally, transforming source code means that a compiling phase is necessary afterwards to regenerate bytecodes. Recompilation is a slow process, much slower than manipulating bytecode; benchmarks of Section 5.3 validate this statement.

Due to the many reasons explained above, a wide variety of tools have been proposed that rely on bytecode transformation. Surprisingly, most of these tools have been made for Java, and we are aware of very few related proposals in the Smalltalk world.

Java and Bytecode Transformation. The Java standard environment only allows for bytecode transformation at load time. At runtime, it is only possible to dynamically generate new classes from scratch, not to modify existing ones. These restrictions have been somehow relaxed in the context of the JVM debugger interface (JDI) [32], but relying on the debugger interface is not reasonable in a production environment. Furthermore, the possibilities of class reloading are strongly limited as, for instance, new members cannot be added to classes. Using load-time transformation in Java also raises a number of subtle issues related to class loaders and the way they define namespaces in Java [33].

Level of Abstraction. The experience gained with Java bytecode transformation tools brings a number of insights that ought to be considered when designing a new framework. The most fundamental one is that of the level of abstraction provided to programmers.

Tools like BCEL [28] and ASM [34] strictly reify bytecode instructions: as a consequence, users have to know the Java bytecode language very well and have to deal with low-level details such as jumps and alternate bytecode instructions (a Java method invocation can be implemented by several bytecode instructions, depending on whether the invoked method is from an interface, is private, etc.).

On the contrary, Javassist [35] and Jinline [25] focus on providing *source code level abstractions*: although the actual transformation is performed on bytecode, the API exposes concepts of the source language. This is highly profitable to end users. In its latest version [29], Javassist even offers a lightweight online compiler so that injected code can be specified as a string of source code. The Javassist compiler supports a number of dedicated metavariables, which can be used to refer to the context in which a piece of code is injected.

As a matter of fact, bytecode-level manipulation is more complex than source-level manipulation because of the many low-level details one needs to deal with. However, working at the bytecode level also makes it possible to express code that is not directly expressable in the source language(s). This dilemma basically motivates the need for both APIs, as is done in Javassist: a high-level API provides source-level abstractions, and a low-level API provides bytecode-level abstractions.

Proposals for Smalltalk. To the best of our knowledge there is no general-purpose bytecode manipulation tool for a Smalltalk dialect. AOStA [36] is a bytecode-to-

bytecode translator that aims at providing higher-level, transparent, type-feedback-driven optimizations. It was not thought to be open to end users for bytecode manipulation [1] . Method wrappers [10] make it possible to wrap a method with before/after code. They are very fast to install and remove, as they do not need to parse bytecode or generate methods, but are not a general-purpose transformation tool. Several extensions actually need more power than just before/after control. AspectS [37] has been recently proposed as an aspect-oriented interface to the reflective capabilities of Smalltalk combined with method wrappers (to implement before/after advices). AspectS is actually a tool that would much profit from BYTESURGEON, as it would significantly raise its expressive power.

2.3 Motivation

From the above, it should be clear that a general-purpose bytecode manipulation tool for Smalltalk is missing. Such a tool ought to provide convenient abstractions to users, both at the source level and bytecode level. BYTESURGEON is precisely such a tool. Beyond its interest for the Smalltalk community, BYTESURGEON also opens the door to a brand new range of experiments with runtime bytecode transformations, since it has none of the limitations of existing Java proposals. For instance, BYTESURGEON makes it possible to analyze concrete issues of fully-dynamic AOP.

3 BYTESURGEON at Work

BYTESURGEON is our library for runtime program transformation in Smalltalk, currently implemented in the Squeak environment. BYTESURGEON complements the reflective abilities of Smalltalk [38] with the possibility to instrument methods, down to method bodies. Smalltalk provides a great deal of structural reflection: the structure of the system is described in itself. Structural reflection can be used to obtain the object representing any language entity. For instance, the global variable Example stands for the class (the object representing the class) Example, and the object describing the compiled method aMethod in class Example is returned by the expression Example>>#aMethod. Dynamically adding instance variables and methods to an existing class is fully supported by any standard Smalltalk environment. However the structural description of a Smalltalk system stops at the level of methods: compiled method cannot be reflected upon. Conversely, BYTESURGEON can be used to do both introspection and intercession on compiled methods.

[1] Actually, BYTESURGEON could profitably use AOStA for its backend, but this study is left as future work.

3.1 Introspecting Method Bodies

Let us first see how BYTESURGEON is used to introspect method bodies. The following code statically counts the number of instructions that occur in all methods of the class Example:

```
InstrCounter reset.
Example instrument: [ :instr | InstrCounter increase ]
```

The instrument: method is implemented in class Behavior. As a parameter it is given a block (of standard Smalltalk code) that takes one argument. This block is an *instrumentation block*: for each instruction within all methods of the class, the instrumentation block is evaluated with a reification of the current instruction as parameter. We will see later what an instruction reification is. For now, suffices to say that for each instruction, a global counter is increased.

There are variants of the instrument: method for each particular language operation: constant, variable access, read and store and message sending. For instance, instrumentSend: only evaluates the instrumentation block upon occurrences of the message send operation. Besides calling the instrumentation method on a class, thereby affecting all its methods, we can call it on a single method:

```
SendMCounter reset.
(Example>>#aMethod) instrumentSend: [ :send | SendMCounter increase ]
```

3.2 Reification of Language Operations

Instructions in a method body are static occurrences of the operations of a language. BYTESURGEON supports message send, access to instance variable and local variables, and constants. The structural model representing language operations is shown on Figure 1 [2]. This structural model is bytecode-based. It does not encode as much information as an AST does, *e.g.,* it is not possible to extract, from an IRSend, the instructions that correspond to the arguments of the send. This is a limitation of bytecode-based transformation against AST-based transformation.

When calling an instrumentation method (*i.e.,* instrument:, instrumentSend:) reification of instructions are built, as instances of the appropriate class in the hierarchy, and passed to the instrumentation block. The instrumentation can then introspect and change them. For instance, the following piece of code prints the selector

[2] The isXXX methods (*e.g.,* isSend) are provided as a convenience to avoid the use of visitors and double dispatch.

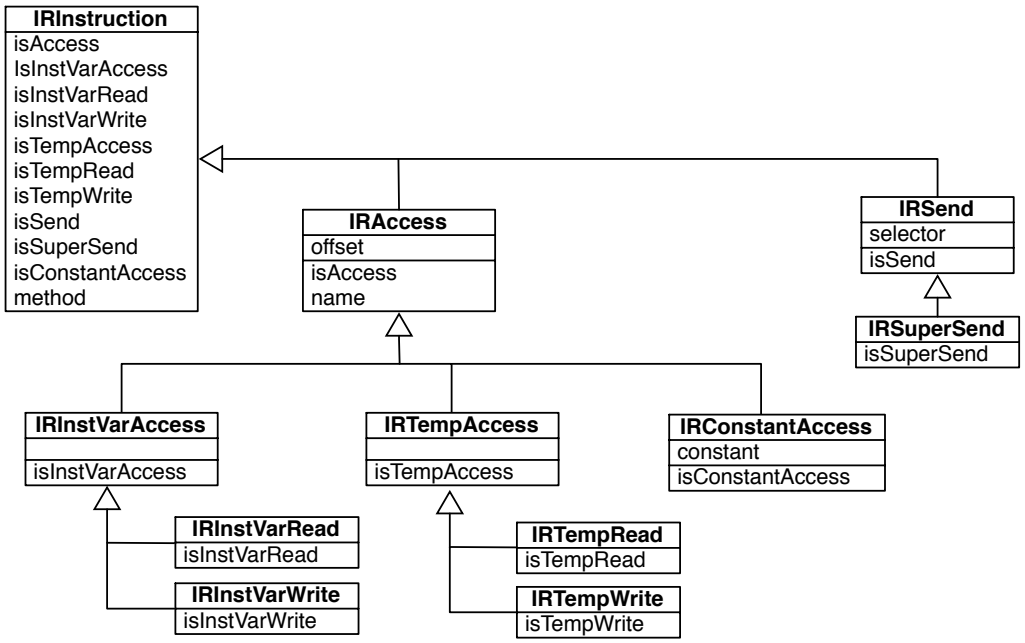

Fig. 1. Structural model of instructions in BYTESURGEON.

of each message send occurring within Example>>#aMethod:

```
(Example>>#aMethod) instrumentSend: [ :send |
                                Transcript show: send selector printString; cr]
```

Method Evaluation. A peculiar language operation is *message receive* (the callee-side equivalent of a message send). Actually, a message receive is realized by two operations: method lookup and method evaluation. Since we are working at the bytecode level, we do not have access to method lookup, only *method evaluation*. Rather than corresponding to a bytecode instruction inside a method body, method evaluation corresponds to a method body as a whole. Since BYTESURGEON treats all language operations in a uniform manner, methods have the same introspection and intercession interface than instructions (*e.g.,* see Section 3.3.3).

3.3 Modifying Method Bodies

BYTESURGEON supports two ways of modifying method bodies: a bytecode-level manner, where the user directly specifies the required transformation in terms of bytecode representations, and a source-level manner, where the transformation is specified with a string of source code. We hereby only present the source-level API. The bytecode-level API is briefly mentioned in Section 4.3.

Similarly to Javassist [29], BYTESURGEON provides an online compiler that makes it possible to specify code to be inserted as a string. The methods to insert code be-

fore, after and instead of an occurrence of a language operation are named respectively insertBefore:, insertAfter: and replace:. They take as argument the source code as a string, which is subsequently compiled by the BYTESURGEON compiler, and the resulting code is inserted at the appropriate position. For instance, the following code inserts a call to the system beeper before each message send occurring within Example>>#aMethod:

```
(Example>>#aMethod) instrumentSend: [ :send | send insertBefore: 'Beeper beep' ]
```

The code string can contain any valid Smalltalk code [3], plus two kinds of special variables: *user-defined variables* to refer to statically-available information, and *metavariables* for runtime information.

3.3.1 Accessing Static Information: User-defined Variables

Statically-known information about an instruction can be used in the construction of the string. For instance, the following example records the name of selector of each message send occurring at runtime:

```
(Example>>#aMethod) instrumentSend: [ :send |
    send insertAfter: 'Logger logSend:' , send selector printString]
```

Here we query the objects describing the message send operations for the name of the message sent. To ease the construction of the string and avoid hard-to-understand string concatenation, BYTESURGEON makes it possible to define custom variables with the syntax $<$: #$variable$$>$, and giving a list of association from variable names to object references [4]:

```
(Example>>#aMethod) instrumentSend: [ :send |
    send insertAfter: 'Logger logSend: <: #sel> ']
        using: { #sel -> send selector }
```

3.3.2 Accessing Runtime Information: Metavariables

The online compiler of BYTESURGEON also supports a number of predefined metavariables that refer to information available at runtime, such as the receiver of a message send (Figure 2). Metavariables are an essential part of the expres-

[3] self, super and thisContext have their usual meaning, knowing that this code will be evaluated in the place where it is inserted.

[4] This is a limited sort of quasi-quoting *a la* Scheme; supporting true quasi-quoting (with no needs to specify manually the associations) is left as future work.

Operation	Metavariable	Description
Message Send/ Method Evaluation	<meta: #arguments>	arguments as an array
	<meta: #argX>	X^{th}argument
	<meta: #sender>	sender object
	<meta: #receiver>	receiver object
	<meta: #result>	returned result (after only)
Temp/InstVar Access	<meta: #value>	value of variable
	<meta: #newvalue>	new value (write only)

Fig. 2. Metavariables supported by BYTESURGEON.

siveness of a good bytecode transformation framework. The exact set of available metavariables depends on both the operation selected –in the case of a message send, metavariables are provided to refer to the sender, the receiver and the arguments– and the transformation to perform –when inserting after, it is possible to access the result–. Metavariables are denoted by the <meta: #$variable$> construct. For instance, the following code replaces each message send with a call to a dispatcher metaobject in charge of the actual method lookup [7, 39]:

```
(Example>>#aMethod) instrumentSend: [ :send | send replace:
  'CustomDispatcher send: <: #selSymbol> to: <meta: #receiver>
                    with: <meta: #arguments> ' ]
  using: { #selSymbol -> send selector printString }
```

The BYTESURGEON online compiler takes care of generating the code to access the runtime information denoted by the metavariables, by adding a preamble before the inlined code. The runtime overhead due to preambles motivated us to maintain a special syntax for metavariables (meta), to raise the attention of users that these variables should be used conscientiously.

3.3.3 Altering Method Evaluation

To support transformation of method evaluation, method objects also support the insertBefore:, insertAfter: and replace: messages. As an example, the following code inserts a trace before each evaluation of a method in Example:

```
Example instrumentMethods:
    [ :m |  m insertBefore: 'Logger logExec: <: #sel> '
              using: { #sel -> m selector } ]
```

The metavariables for method evaluation are the same as for message sending (see Figure 2). The following example uses a metavariable to access the method evaluation result:

```
Example instrumentMethods:
  [ :m |  m insertAfter: 'Logger logExec: <: #sel> result: <meta: #result> '
          using: { #sel –>   m selector } ]
```

4 Inside BYTESURGEON

We now give an overview of the implementation of BYTESURGEON, in particular the relation with the closure compiler and the transformation process. The low-level transformation API is also discussed.

4.1 Squeak

BYTESURGEON is currently implemented in Squeak [40], an open source implementation of Smalltalk-80 [41]. Squeak is based on a virtual machine that interprets bytecodes. During a normal compilation phase, method source code is scanned and parsed, an abstract syntax tree (AST) is created and bytecodes are generated for the corresponding methods (Figure 3).

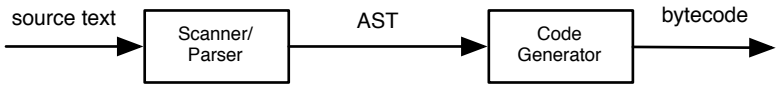

Fig. 3. The standard Smalltalk to bytecode compiler.

To implement BYTESURGEON in Squeak we could have directly work on bytecode. However, rewriting bytecode is tedious and error-prone for several reasons: the bytecode vocabulary is low-level, jumps have to be calculated by hand, the expression of the context where bytecodes should be inserted is limited. Even simple modifications are surprisingly tedious to manage. Fortunately, a new compiler for Squeak, the *closure compiler*, has been recently proposed which offers a better intermediate bytecode representation.

4.2 The Closure Compiler and its Intermediate Representation

The closure compiler [42] relies on a more complex bytecode generation step (Figure 4): first an *Intermediate Representation (IR)* is created; then the IR is used to generate the real bytecode (the raw numbers).

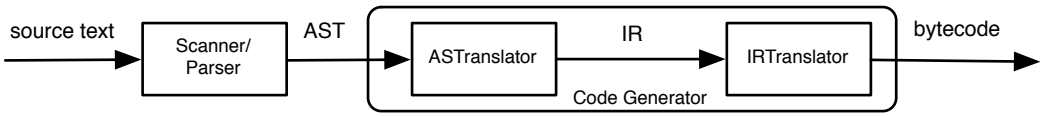

Fig. 4. The closure compiler.

The IR is a high-level representation of bytecode, abstracting away specific details: jumps are encoded in a graph structure, sequences of bytecode-nodes form a basic block, and jump-bytecodes concatenate these blocks to encode control flow. The main goal of IR is to abstract from specific bytecode encodings: for instance, although the bytecode for a program in Squeak is encoded differently than in VisualWorks, their IR is identical. Using IR therefore makes the porting to other bytecode sets simple.

The closure compiler has a counterpart, the decompiler, which converts bytecode back to text. Here, the whole process works backwards: from bytecode to IR, from IR to AST, and finally from AST to text.

As motivated in Section 2.2, BYTESURGEON ought to offer adequate abstractions for both bytecode-level and source-level transformations. The IR of the closure compiler actually represents an excellent alternative for working at the bytecode level: it makes it possible to express code that is not directly obtainable from Smalltalk source code, while abstracting away many details.

All classes reifying instructions (recall Figure 1) are from the closure compiler IR. The low-level transformation API of BYTESURGEON is based on these classes. In addition to the classes reifying instructions which correspond to language operations, the IR includes classes reifying bytecode-only instructions: IRPop, IRDup, IRJump, IRReturn, etc.

4.3 Low-level Transformation API

In Section 3, we have used the high-level API of BYTESURGEON to specify transformations giving a string of source code, which may contain metavariables to access dynamic information. The description of the new code to be inlined can also be done by directly editing the instruction objects for the IR hierarchy. In the following example, the selector of all sends of the message oldMessage:with: are replaced by sends of the message newMessage:with:, by using the selector: accessor of an IRSend object:

```
(Example>>#aMeth) instrumentSend: [ :send | send selector = #oldMessage:with:
        ifTrue: [ send selector: #newMessage:with: ] ].
```

The IRInstruction class can also be used as a factory to produce new objects describing bytecode. These objects can be used in replacement of the original instruction or be inlined before or after it. An alternative implementation of the code above is:

```
(Example>>#aMeth) instrumentSend: [ :send | send selector = #oldMessage:with:
        ifTrue: [ send replace: (IRInstruction send: #newMessage:with:)] ].
```

This implementation replaces the message send bytecode by a new one having a different selector. IRInstruction send: #newMessage:with: returns an object that describes a message send bytecode.

Specifying the transformation at the bytecode-level makes it possible to express constructs that are impossible at the level of the Smalltalk language, and to easily specify transformations that are more complex to express with the source-level API. For instance, using the source-level API to change the selector of a message send, as done above, is done as follows:

```
(Example>>#aMeth) instrumentSend: [ :send | send selector = #oldMessage:with:
    ifTrue: [ send replace: '<meta: #receiver> perform: #newMessage:with:
                with: <meta: #arguments> ' ] ].
```

Apart from being slightly more verbose and relying on the use of the reflective message sending perform:with:, this approach requires the use of metavariables, which are more costly due to the associated preambles that needs to be generated (as shown in Section 4.4). Conversely, the low-level API makes it possible to do this transformation directly, without requiring runtime reification.

4.4 Implementation of Metavariables

When BYTESURGEON instruments a method, the bytecode-to-IR part of the closure compiler generates the IR objects that are passed to the instrumentation block specified by the user. If the source-level API is used, then the code to be inserted is preprocessed to generate the IR nodes and to handle the metavariables, if any. For metavariables, a preamble code is generated to ensure that the expected values will be on the stack. Then, the preamble and code are inserted into the IR of the method. Finally, the IR-to-bytecode part of the closure compiler generates raw bytecodes and replaces the old method with the new, transformed version.

In the following we explain the implementation of metavariables which reify runtime information. Let us consider the reification of the receiver of a message send.

Preambles. Squeak uses a stack-based bytecode, so all parameters for a message

send are pushed on the stack before the send bytecode is executed: first the receiver, and then the arguments. For instance, the bytecode for the expression 3 + 4 is as follows:

```
77 pushConstant: 3
20 pushConstant: 4
B0 send: +
7C returnTop
```

Consider that we now want to provide access to the receiver (3) via a metavariable:

```
(Example>>#method) instrumentSend: [:send |
                send insertBefore: 'Transcript show: <meta: #receiver> as-
String'].
```

To support metavariables, we need to add bytecode to store the necessary values, by popping them from the stack and storing them in additional temporary variables. In our example, we need the receiver. Since the receiver is deep in the stack, below the arguments, we also need to store the arguments in temporary variables, to be able to access them afterwards. In the case of before/after, it is also necessary to rebuild the stack. The resulting bytecode for our example is as follows:

```
22 pushConstant: 3
23 pushConstant: 4
68 popIntoTemp: 0                "put argument in temp 0"
69 popIntoTemp: 1                "put receiver in temp 1"
24 pushLit: ##Transcript         "start of inserted code"
11 pushTemp: 1                   "push receiver for printing"
D5 send: asString
E6 send: show:
87 pop                           "end of inserted code"
11 pushTemp: 1                   "rebuild the stack"
10 pushTemp: 0
B0 send: +                       "original code"
7C returnTop
```

To access all arguments as an array, the compiler generates code to create the array instance, to add arguments to it, and to store the array in a temporary variable.

For performance and space reasons, preamble generation needs to be optimized. First, the compiler only generates code for the metavariables that are effectively used in the inlined code. For instance, if access to the arguments is not needed, then the array creation is avoided. The second important optimization is to reuse

temporary variables. Indeed, there are potentially many operations for which we need to generate a preamble, in a single method. If we used new temporary variables for each, we would soon run out of temporary variables (Squeak imposes a limit of 256 temporary variables per method). Therefore, BYTESURGEON remembers the original number of temporary variables and reuses the variables added for each preamble. This information is saved inside the compiled method object, so that reuse of variables works even if instrument: is executed several times on the same method.

Inlining code. Once the preamble is added, the code to inline can be inserted. First, the BYTESURGEON compiler generates the IR for the new code. For metavariables, the compiler generates code that loads the corresponding temporary variables. The generated IR instructions are then added to the original IR of the method. If necessary, jump targets are adjusted and basic blocks renumbered. The new method IR is then given to the closure compiler, which generates the final raw bytecodes and installs the new method.

5 Validation

We now validate the interest of BYTESURGEON by showing how easy it is to implement two language extensions: method wrappers [10] and a simple runtime MOP for controlling accesses to instance variables. Section 5.3 completes this validation by reporting on performance measurements.

5.1 Method Wrappers

Method wrappers [10] wrap a method with before/after behavior. Wrapping a method is implemented by swapping out the compiled method by another one, valueWith-Receiver:arguments: that calls the before method, then the original method, and finally the after method [5] :

```
MethodWrapper>>valueWithReceiver: anObject arguments: args
    self beforeMethod.
    ^ [clientMethod valueWithReceiver: anObject arguments: args]
      ensure: [self afterMethod]
```

[5] At the time of this writing, BYTESURGEON does not yet support exception handlers, so we actually implemented a simplified version where the after method is just inlined at the end of the method.

The BSMethodWrapper class contains the logic to install an instance of itself as a method wrapper, with empty before/after methods.

To define a wrapper, a subclass should be created, specifying the before/after methods. For instance, class CountingMethodWrapper wraps a method to count invocation of calls to a given method:

```
BSMethodWrapper subclass: #CountingMethodWrapper
  instanceVariableNames: 'count'...

CountingMethodWrapper >>beforeMethod
  self count: self count + 1
```

To count the invocations on a method, we install the wrapper:

```
wrapper  := CountingMethodWrapper on: #aMethod inClass: Example.
wrapper install.
```

The installation of a method wrapper consists in first decompiling the before/after methods to IR (ir), stripping the return at the end (strip), then replacing all self references to refer to the wrapper (replaceSelf:), and finally inlining the before/after methods (insertBefore:after:):

```
BSMethodWrapper>>inlineBeforeAfter
    | before after |
    before := (self class lookupSelector: #beforeMethod) ir strip.
    after := (self class lookupSelector: #afterMethod) ir strip.

    self replaceSelf: before. self replaceSelf: after.
   self method insertBefore: before startSequence after: after startSequence.

BSMethodWrapper>>replaceSelf: ir  "replace self with pointer to me"
 ^ ir allInstructions do: [:instr | instr isSelf ifTrue: [
                              instr replaceWith: (IRInstruction pushLiteral: self)]].
```

As we can see, method wrappers are straightforward to implement with BYTESURGEON. The complete implementation included in the distribution consists of 41 lines of code, with comments. This implementation of method wrapper should only serve as an example of use of BYTESURGEON, it is not meant to be a replacement yet since not all features of method wrappers are supported. Furthermore, as illustrated in Section 5.3, standard method wrappers and BYTESURGEON method wrappers have different performance profiles.

We now show how to implement a small runtime MOP for controlling accesses to instance variables. A metaobject can be associated to a class, and upon accesses to instance variables of objects from the class, it gets control via either its instVarRead:in: method (if it is a read access) or its instVarWrite:in:value: method (if it is a write access). For instance, the following TraceMO simply outputs what is happening to the transcript and then performs the standard action, *i.e.*, returning the instance variable value, or storing the new value:

```
TraceMO>>instVarRead: name in: object
    | val |
    val := object instVarNamed: name.
    Transcript show: 'var read: ', val printString; cr.
    ^val.

TraceMO>>instVarStore: name in: object value: newVal
     Transcript show: 'var store: ', newVal printString; cr.
     ^object instVarNamed:  name put: newVal.
```

This metaobject can be installed on class Point as follows:

```
   MOP install: TraceMO new on: Point
```

The MOP>>install method uses BYTESURGEON to replace the bytecodes that read or store instance variables with calls to the metaobject (*aka.* hooks):

```
MOP class >>install: mop on: aClass
  | dict |
  dict := Dictionary newFrom: #mo -> mop.
  aClass instrumentInstVarAccess: [:instr |
     dict at: #name put: instr varname.
     instr isRead
       ifTrue: [instr replace: '<: #mo> instVarRead: <: #name> in: self'
                    using: dict]
       ifFalse: [instr replace: '<: #mo> instVarStore: <: #name> in: self
                                 value: <meta: #newvalue> '
                    using: dict] ].
```

The dict dictionary is used to hold the reference to the metaobject controlling accesses, and for each access instruction, the name of the variable is put in it. This makes it possible to use user-defined variables when specifying the transformation.

Furthermore, since BYTESURGEON supports runtime bytecode manipulation, we are able to completely *uninstall* hooks when needed:

```
MOP uninstall: MOExample.
```

Of course, this simple MOP is not complete: if methods are changed (recompiled), the MOP is removed, there is no way to compose multiple metaobjects on the same class, it is not possible to associate different metaobjects to different instances, etc. But the basic features are there: a MOP for instance variable accesses that can be installed and retracted at runtime –and completely implemented in *less than 10 lines*–.

5.3 Benchmarks

We now report on several preliminary benchmarks [6] we have performed to evaluate the efficiency of BYTESURGEON. First, we report on transformation vs. compilation costs, and then study the performance of the standard implementation of method wrappers with that based on BYTESURGEON.

Transformation performance. One of the reasons for editing bytecode instead of source is performance. To verify this claim, we have carried out a simple set of benchmarks, in which we compare the time to compile some code with both the standard compiler of Squeak and the new compiler (closure compiler), and the time taken by BYTESURGEON to transform all instructions in the code with an empty block. Hence what we actually measure for BYTESURGEON is the time it takes to decompile methods to IR, execute the block for each instruction (which does nothing), generate a new identical method and install it.

The first benchmark is applied to the Object class:

```
"Test compilers"
[Object compileAll] timeToRun

"Test ByteSurgeon"
[Object instrument: [:inst | self ]] timeToRun
```

Class Object contains 429 methods, amounting to 2344 lines of code. We did the same experiment on a larger code base: the whole hierarchy of collection classes. This hierarchy consists of 76 classes, 2231 methods, summing up to 15783 lines of code. The benchmark is run as:

[6] Machine used: Apple PowerBook 1.5Ghz, Squeak 3.8

	Object		Collections	
	time (ms)	factor	time (ms)	factor
BYTESURGEON	661	1	4817	1
standard compiler	1232	1.86	9760	2.03
closure compiler	3673	5.55	33611	6.98

Fig. 5. Comparing compilation and transformation times.

```
"Test compilers"
[Collection allSubclasses do: [ :c | c compileAll ]] timeToRun

"Test ByteSurgeon"
[Collection allSubclasses do: [ :c | c instrument: [ :inst | self ]]] timeToRun
```

The results of both benchmarks are presented in Figure 5. As expected, BYTESURGEON performs very well. The highly optimized standard compiler is approximately twice slower than BYTESURGEON, while the new compiler, which is much easier to reuse and extend but less optimized, is around 6 times slower.

Method wrapper performance. We now compare the performance of the standard implementation of method wrappers with that based on BYTESURGEON. We compare both installation (transformation) time and execution time.

The test consists of a simple before/after counter manipulation wrapping a straightforward method:

```
Bench>>run          beforeMethod        afterMethod
    ^ 3+4.           BCounter inc        BCounter inc
```

The benchmark of the installation/uninstallation is run as follows:

```
[1000 timesRepeat: [
  w := TestMethodWrapper on: #run inClass: Bench.
  w install. w uninstall]] timeToRun
```

The runtime performance of both implementations is compared to that of method that directly implements the wrapper:

```
Bench>>run
   | t |
   BCounter inc.
   t := 3+4.
```

Method Wrapper	Installation		Runtime	
implementation	time (ms)	factor	time (ms)	factor
Hand-coded	–	–	1253	1
Standard	603	1	6732	5.37
BYTESURGEON	3710	6.01	1222	0.98

Fig. 6. Comparing installation and runtime performance of method wrapper implementations.

```
BCounter inc.
^t.
```

To be fair in our evaluation, we changed the execution semantics of standard method wrappers, so that they do not wrap the after in an exception handler, but rather inline both before and after methods. The benchmark for both cases is run as follows:

```
[1000000 timesRepeat: [Bench new run]] timeToRun
```

The results of the benchmarks (Figure 6) show that BYTESURGEON is slower for installing wrappers. This was expected because method wrappers actually simply swap the wrapped compiled method with the wrapper one, while BYTESURGEON actually modifies the original method. The other side of the coin is that BYTESURGEON-based method wrappers are much more efficient at runtime. Standard method wrappers are 3.5 times slower than the hand-coded version, while the BYTESURGEON implementation is as fast as the hand-coded version. The slight enhancement that can be observed comes from the fact that, in the considered case, BYTESURGEON does not need to use a temporary variable to store the return value, it just uses the stack.

6 Conclusion and Future Work

We have presented BYTESURGEON, an efficient library for runtime bytecode manipulation in Smalltalk, implemented in Squeak. We have shown:

- APIs for specifying transformations that allow users to control the tradeoff between expressiveness and performance for the code to be inlined: BYTESURGEON users can either specify Smalltalk code with metavariables or specify the code at the bytecode level.
- the expressiveness of BYTESURGEON by showing how well-known language extensions are concisely expressed, and reported on preliminary benchmarks validating our efficiency claim.

- the runtime capabilities of BYTESURGEON with a simple MOP that can be dynamically installed and retracted. Such runtime changes are not feasible in a static system like Java without changing the virtual machine.

Future work can be dividing in two directions: the first is to continue improving BYTESURGEON as such, and the second consists in using BYTESURGEON in a number of projects that will directly benefit from its features. Of course, both tracks mutually benefit from each other.

Regarding BYTESURGEON itself, there is a number of features that are being discussed at this time. In particular, BYTESURGEON should be extended with support for exception handling. It is also appealing to offer a kind of proceed instruction to trigger the execution of a replaced operation occurrence from inside the meta-computation. Another direction to explore is that of the abstraction layer used to describe a method. As of now we use a bytecode representation, but it would be interesting to explore the direct use of abstract syntax trees at this level. The choice between AST and bytecode presents a tradeoff between performance and expressiveness: decompiling to AST and code-generation will be slower than using the the bytecode-level abstractions of the IR, but in turn we gain a lot in expressiveness and ease of use, since AST is more structured than IR. We plan to explore these tradeoffs in the future.

As regards applications of BYTESURGEON in other projects, the perspectives are manifold. We plan to use BYTESURGEON for code annotation to collect runtime traces of program execution to support *omniscient debugging* [11]. *Reflex* is a system based on bytecode transformation providing partial behavioral reflection in Java [43]. It has recently evolved to a versatile kernel for multi-language AOP [44], easing the implementation of (domain-specific) aspect languages and providing support for the detection and resolution of aspect interactions. The on-going *Geppetto* project aims at exploring the possibilities offered by an implementation of Reflex in Squeak, using BYTESURGEON, enjoying the flexibility of true runtime code transformation.

Acknowledgements. We thank David Röthlisberger and the anonymous reviewers for their comments.

References

[1] R. Keller, U. Hölzle, Binary component adaptation, in: ECOOP'98, LNCS 1445, 1998, pp. 307–340.

[2] D. L. Parnas, On the criteria to be used in decomposing systems into modules, CACM 15 (12) (1972) 1053–1058.

[3] R. Stroud, Z. Wue, Using metaobject protocols to satisfy non-functional requirements, in: Advances in Object-Oriented Metalevel Architectures and Reflection, CRC Press, 1996, pp. 31–52.

[4] É. Tanter, J. Piquer, Managing references upon object migration: Applying separation of concerns, in: Proceedings of the XXI International Conference of the Chilean Computer Science Society (SCCC 2001) (jan 2001).

[5] J. McAffer, Meta-level architecture support for distributed objects, in: Proceesings of the Fourth International Workshop on Object-Orientation in Operating Systems, 1995., 1995, pp. 232–241.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: M. Aksit, S. Matsuoka (Eds.), Proceedings ECOOP '97, Vol. 1241 of LNCS, Springer-Verlag, Jyvaskyla, Finland, 1997, pp. 220–242.

[7] S. Ducasse, Evaluating message passing control techniques in Smalltalk, Journal of Object-Oriented Programming (JOOP) 12 (6) (1999) 39–44.

[8] J. H. Heinz-Dieter Bocker, What tracers are made of, in: Proceedings of OOPSLA/ECOOP '90, 1990, pp. 89–99.

[9] F. Pachet, F. Wolinski, S. Giroux, Spying as an Object-Oriented Programming Paradigm, in: Proceedings of TOOLS EUROPE '93, 1993, pp. 109–118.

[10] J. Brant, B. Foote, R. Johnson, D. Roberts, Wrappers to the Rescue, in: Proceedings ECOOP '98, Vol. 1445 of LNCS, Springer-Verlag, 1998, pp. 396–417.

[11] B. Lewis, Debugging backwards in time, in: Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003) (oct 2003).

[12] A. J. Ko, B. A. Myers, Designing the whyline: a debugging interface for asking questions about program behavior, in: Proceedings of ACM CHI 2004 Conference on Human Factors in Computing Systems, Vol. 1, 2004, pp. 151–158.

[13] A. H. Borning, D. H. Ingalls, Multiple inheritance in Smalltalk-80, in: Proceedings at the National Conference on AI, Pittsburgh, PA, 1982, pp. 234–237.

[14] B. Garbinato, R. Guerraoui, K. R. Mazouni, Distributed programming in GARF, in: R. Guerraoui, O. Nierstrasz, M. Riveill (Eds.), Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, Vol. 791 of LNCS, Springer-Verlag, 1994, pp. 225–239.

[15] J. K. Bennett, The design and implementation of distributed Smalltalk, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 318–330.

[16] P. L. McCullough, Transparent forwarding: First steps, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 331–341.

[17] K. Beck, Instance specific behavior: Digitalk implementation and the deep meaning of it all, Smalltalk Report 2(7).

[18] J.-P. Briot, Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment, in: S. Cook (Ed.), Proceedings ECOOP '89, Cambridge University Press, Nottingham, 1989, pp. 109–129.

[19] Y. Yokote, M. Tokoro, Experience and evolution of ConcurrentSmalltalk, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 406–415.

[20] G. A. Pascoe, Encapsulators: A new software paradigm in Smalltalk-80, in: Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, 1986, pp. 341–346.

[21] B. Foote, R. E. Johnson, Reflective facilities in Smalltalk-80, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 327–336.

[22] J. McAffer, Meta-level programming with coda, in: W. Olthoff (Ed.), Proceedings ECOOP '95, Vol. 952 of LNCS, Springer-Verlag, Aarhus, Denmark, 1995, pp. 190–214.

[23] W. R. LaLonde, M. V. Gulik, Building a backtracking facility in Smalltalk without kernel support, in: Proceedings OOPSLA '88, ACM SIGPLAN Notices, Vol. 23, 1988, pp. 105–122.

[24] G. Kiczales, J. des Rivières, D. G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[25] É. Tanter, M. Ségura-Devillechaise, J. Noyé, J. Piquer, Altering Java semantics via bytecode manipulation, in: Proceedings of GPCE'02, Vol. 2487 of LNCS, Springer-Verlag, 2002, pp. 283–89.

[26] M. Tatsubori, S. Chiba, M.-O. Killijian, K. Itano, OpenJava: A class-based macro system for java, in: 1st OOPSLA Workshop on Reflection and Software Engineering, Vol. 1826 of LNCS, Springer Verlag, 2000, pp. 117–133.

[27] J. Bachrach, K. Playford, The Java Syntactic Extender (JSE), Proceedings of OOPSLA '01, ACM SIGPLAN Notices 36 (11) (2001) 31–42.

[28] M. Dahm, Byte code engineering, in: Proceedings of JIT '99, Düsseldorf, Deutschland, 1999, pp. 267–277.

[29] S. Chiba, M. Nishizawa, An easy-to-use toolkit for efficient Java bytecode translators, in: Proceedings of GPCE'03, Vol. 2830 of LNCS, 2003, pp. 364–376.

[30] D. A. Smith, A. Kay, A. Raab, D. P. Reed, Croquet, A Collaboration System Architecture, in: Proceedings of the First Conference on Creating, Connecting and Collaborating through Computing (2003).

[31] Jython, http://www.jython.org/.

[32] Java debug interface (jdi), http://java.sun.com/j2se/1.4.2/docs/jguide/jpda/jarchitecture.html.

[33] S. Liang, G. Bracha, Dynamic class loading in the Java virtual machine, in: Proceedings of OOPSLA '98, ACM SIGPLAN Notices, 1998, pp. 36–44.

[34] E. Bruneton, R. Lenglet, T. Coupaye, ASM: A code manipulation tool to implement adaptable systems, in: Proceedings of Adaptable and extensible component systems (nov 2002).

[35] S. Chiba, Load-time structural reflection in Java, in: Proceedings of ECOOP 2000, Vol. 1850 of LNCS, 2000, pp. 313–336.

[36] E. Miranda, A Sketch for an Adaptive Optimizer for Smalltalk written in Smalltalk, unpublished (2002).

[37] R. Hirschfeld, AspectS – Aspect-Oriented Programming with Squeak, in: M. Aksit, M. Mezini, R. Unland (Eds.), Objects, Components, Architectures, Services, and Applications for a Networked World, no. 2591 in LNCS, Springer, 2003, pp. 216–232.

[38] F. Rivard, Smalltalk : a Reflective Language, in: Proceedings of REFLECTION '96, 1996, pp. 21–38.

[39] J. Ferber, Computational reflection in class-based object-oriented languages, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 317–326.

[40] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, A practical Smalltalk written in itself, in: Proceedings OOPSLA '97, ACM SIGPLAN Notices, ACM Press, 1997, pp. 318–326.

[41] A. Goldberg, D. Robson, Smalltalk 80: the Language and its Implementation, Addison Wesley, Reading, Mass., 1983.

[42] A. Hannan, Squeak Closure Compiler, http://minnow.cc.gatech.edu/squeak/ClosureCompiler.

[43] É. Tanter, J. Noyé, D. Caromel, P. Cointe, Partial behavioral reflection: Spatial and temporal selection of reification, in: Proceedings of OOPSLA '03, ACM SIGPLAN Notices, 2003, pp. 27–46.

[44] É. Tanter, J. Noyé, A versatile kernel for multi-language AOP, in: Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005), Vol. 3676 of LNCS, Tallin, Estonia, 2005.

Towards a Taxonomy of SUnit Tests ⋆

Markus Gälli [a] Michele Lanza [b] Oscar Nierstrasz [a]

[a] *Software Composition Group*
Institut für Informatik und angewandte Mathematik
Universität Bern, Switzerland

[b] *Faculty of Informatics*
University of Lugano, Switzerland

Abstract

Although unit testing has gained popularity in recent years, the style and granularity of individual unit tests may vary wildly. This can make it difficult for a developer to understand which methods are tested by which tests, to what degree they are tested, what to take into account while refactoring code and tests, and to assess the value of an existing test. We have manually categorized the test base of an existing object-oriented system in order to derive a first taxonomy of unit tests. We have then developed some simple tools to semi-automatically categorize tests according to this taxonomy, and applied these tools to two case studies. As it turns out, the vast majority of unit tests focus on a single method, which should make it easier to associate tests more tightly to the methods under test. In this paper we motivate and present our taxonomy, we describe the results of our case studies, and we present our approach to semi-automatic unit test categorization.

Key words: unit testing, taxonomy, reverse engineering

1 Introduction

XUnit [1] in its various forms (JUnit for Java, SUnit for Smalltalk, etc.) is a widely-used open-source unit testing framework. It has been ported to most object-oriented programming languages and is integrated in many common IDEs such as Eclipse.

⋆ We thank Stéphane Ducasse for his helpful comments and gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02).

Email addresses: `gaelli@iam.unibe.ch` (Markus Gälli), `michele.lanza@unisi.ch` (Michele Lanza), `oscar@unibe.ch` (Oscar Nierstrasz).

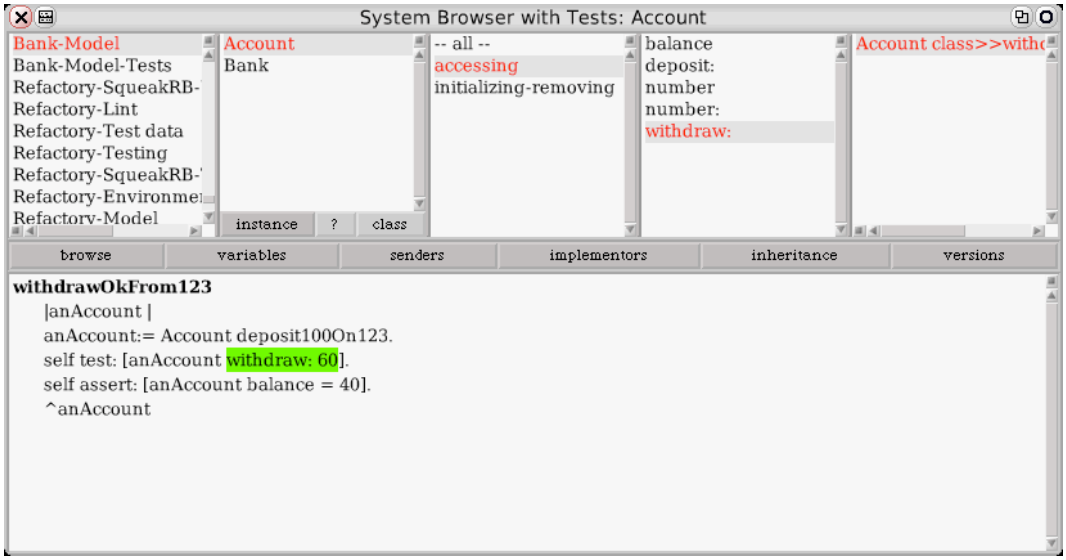

Fig. 1. An enhanced class browser shows methods and their one-method tests site by site. Note that the test returns its result, thus enabling other unit tests to reuse it. We thus store tests like other factory methods on the class site.

Although these development environments help developers to navigate between related methods in a complex software system, they offer only limited help in relating methods and the unit tests that test them.

Our hypothesis is that a majority of unit tests focus on single methods. We call these dedicated unit tests *one-method commands*. If our hypothesis is valid, then we could help the developer in several ways to write and evolve methods together with their tests:

- *Tighter integration of tests and methods in class browsers*. Each *one-method command* could be displayed close to its method, and document a quality-approved usage of the method. (See Figure 1) It then would be also clear if a method has a *dedicated* test case or not. The developer would not have to switch windows for developing tests or methods as they could be naturally displayed site by site.
- *Test case selection*. All *one-method commands* could be executed as soon as their focused method has been changed.
- *Concrete Typing*. The set of tested concrete types of the receiver, parameters and result of the method under test are deducible by executing an instrumented version of its *one-method commands*. Thus *one-method commands* remove the burden of a test-first-driven development of providing the types in a statically typed language or deducing them in a dynamically typed language.
- *Test case refactoring*. If a method is deleted, its corresponding test method could be deleted immediately too. Renaming a method would not break the brittle naming convention anymore, which is currently the only link between a method and its unit tests. Adding a parameter to a method could be automatically mirrored

by adding a factory to its according test [1] .

In order to validate our hypothesis we have:

- Developed an initial taxonomy of unit tests by carrying out an empirical study of a substantial collection of tests produced by a community of developers.
- Implemented some lightweight tools to automatically classify certain tests into categories offered by the taxonomy.
- Conducted case studies to validate the generality of the taxonomy.

Our manual experiment supports the hypothesis that a significant portion of test cases have an implicit one-to-one relationship to a method under test or are decomposable into *one-method commands*. Although it is difficult to identify a general algorithm to distinguish this kind of test, our initial heuristics to automate this endeavor succeed in identifying 50% of one-to-one tests without resulting in any false positives.

Structure of the article. In Section 2 we define some basic terms. In Section 3 we present the taxonomy derived from our manual case study. In Section 4 we describe some simple heuristics for mapping unit tests to the taxonomy, and we describe the results of applying these heuristics to two case studies. In Section 5 we discuss some of the problems and difficulties encountered. Section 6 briefly outlines related work. In Section 7 we conclude and outline future work.

2 Basic Definitions

We first introduce some basic terminology, on which our taxonomy builds on.

Assertion: An *assertion* is a method that evaluates a (side-effect free) Boolean expression, and throws an exception if the assertion fails. Unit test assertions usually focus on specific instances whereas assertions of *Design By Contract* are used in post-conditions and are more general.

Package: We assume the existence of a mechanism for grouping and naming a set of classes and methods. In the case of Java this would be packages; in the case of Smalltalk we use class categories as the smallest common denominator of several Smalltalk dialects. We call these groups *packages*.

Command: Every XUnit Test is a *command* [3], which is a parameter-free method whose receiver can be automatically created. The XUnit Test can thus be automatically executed.

[1] Further refactorings [2], which have to be carried out in parallel for the test code and the code under test would be easier too, but this is subject to further research.

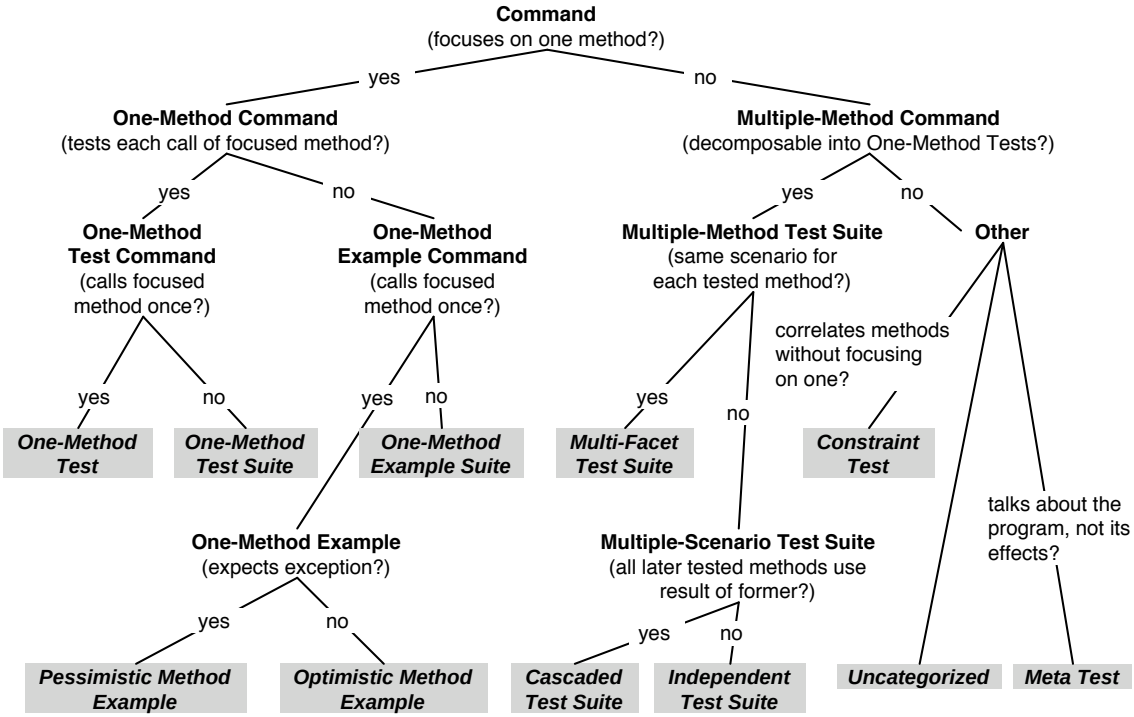

Fig. 2. Taxonomy of unit tests. Nodes are gray and denote concrete occurrences of unit tests.

The command receiver in the case of a XUnit test case can be constructed automatically, *e.g.*, new MyTestCase(myTestSelector). The whole command then looks like:

```
(new MyTestCase(myTestSelector)).run()
```

Test package: A *test package* is a package which includes a set of commands.

Package under Test: If a test package tests another package, we call this other package the *package under test*, which may be identified either implicitly by means of naming conventions, or explicitly by means of a dependency declaration.

Candidate method: A *candidate method* is a method of the package under test.

Focuses on one method: We say that a command *focuses on one method*, if it tests the result or side effects of *one* specific method and not the result or side effects of several methods.

3 A Taxonomy of Unit Tests

Initial case study. We derived the taxonomy by manually categorizing 982 unit tests of the Squeak [4] base system [2] . Squeak is a feature-rich, open source implementation of the Smalltalk programming language written in itself and by many developers. It includes network- and 2D/3D-graphics support, an integrated development environment, and a constructivist learning environment for children.

The tests were written by at least 26 different developers. One of the test developers developed 36% of the test cases, two more developed a further 34%, and yet another six developers produced another 19% of tests. Each of the other developers produced less than 3% of the tests. We defined the taxonomy depicted in Figure 2 by iteratively grouping tests into categories and refining the classification criteria. Our manual categorization yielded a distribution of the categories shown in Figure 3.

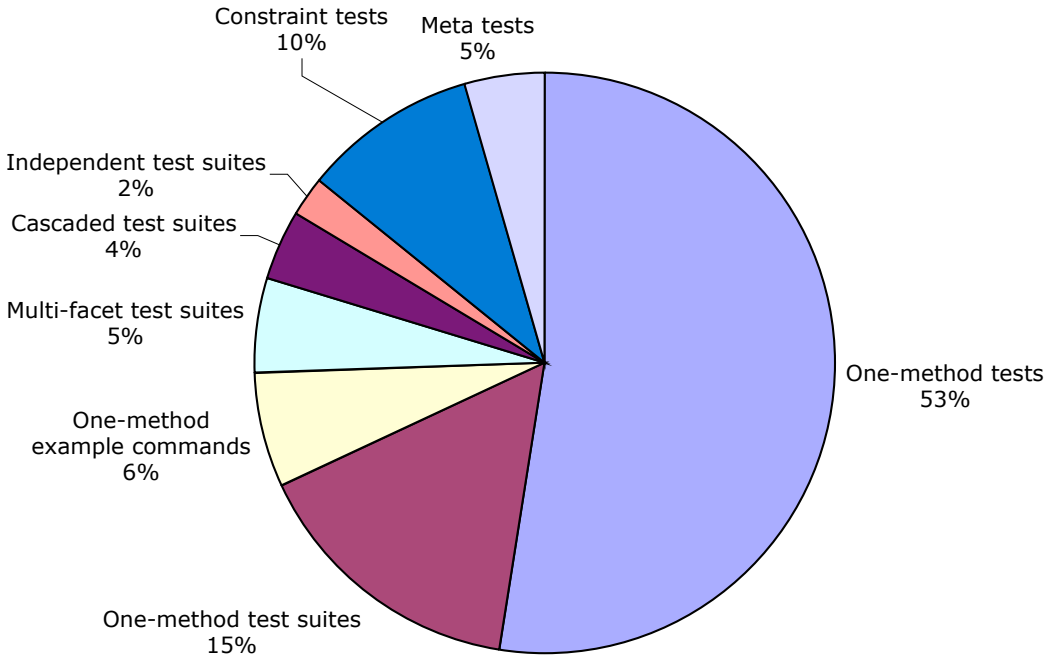

Fig. 3. Manual classification of unit tests for the base Squeak system

We now describe and motivate each of the unit test categories in the taxonomy. For each node of our taxonomy we present a real world example found in the Squeak unit tests [3] .

We divide our taxonomy tree into two subtrees (Figure 2): (1) *One-method commands*, which are *commands* that focus on single methods, and (2) *multiple-method*

[2] Version 3.7 beta update 5878, available at `http://www.squeak.org`
[3] For a short introduction to the Smalltalk syntax see the appendix.

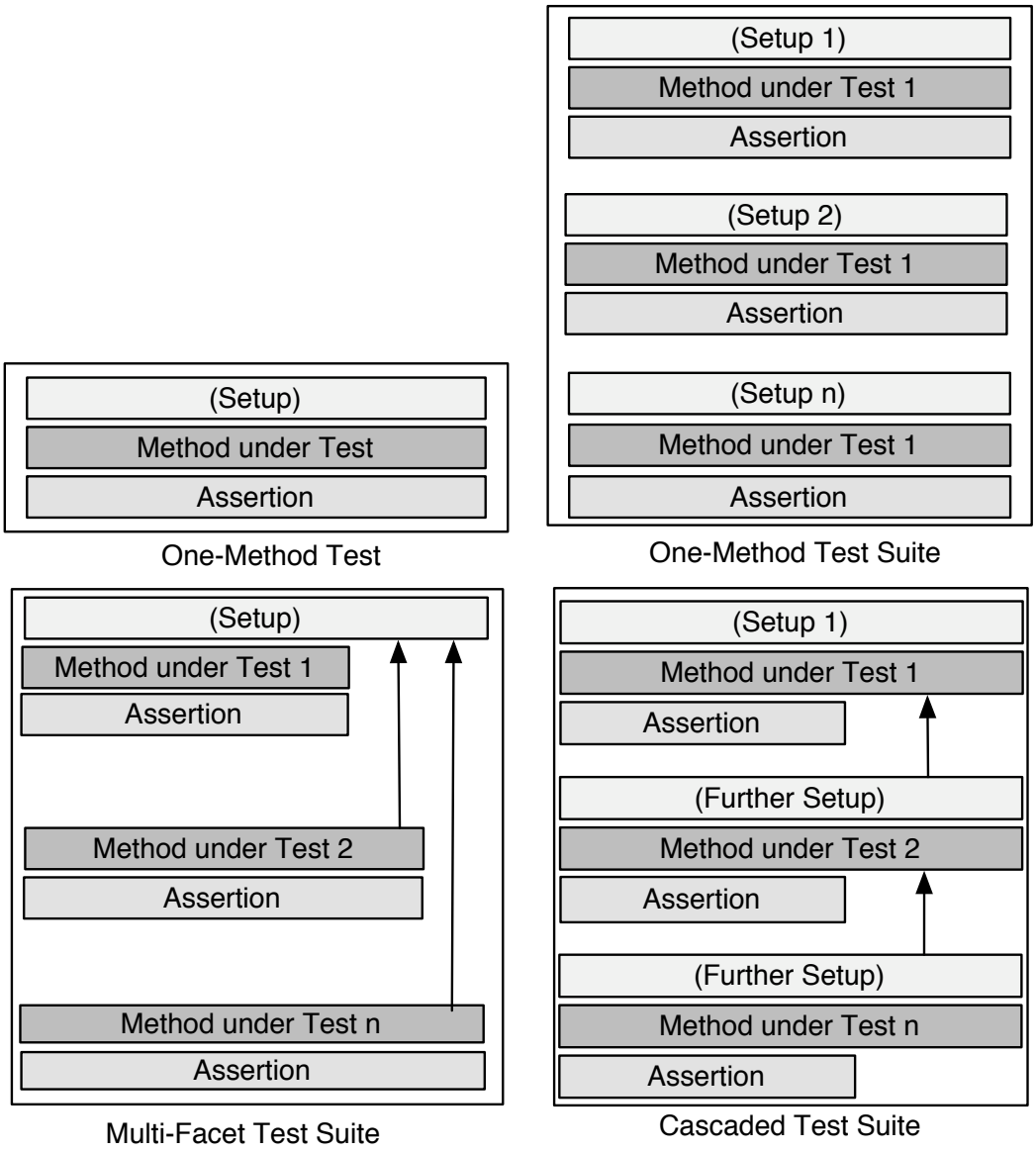

Fig. 4. One-method test suites, multi-facet test suites and cascaded test-suites are decomposable into one-method tests.

commands, which do not focus on a single method. We divided each of these subtrees into two further subtrees, which we will present in the following subsections.

3.1 One-method test commands

A *one-method test comand* is a *one-method command* which has assertions testing the outcome of *each call* of the method under test.

3.1.1 One-method tests

If it tests the outcome of exactly one call of a method under test, we call it a *one-method test*. In the example below the method Week class≫indexOfDay: would be the method under test, and only called once:

```
YearMonthWeekTest≫testIndexOfDay
  self assert: (Week indexOfDay: 'Friday') = 6.
```

3.1.2 One-method test suites

On the other hand a *one-method test suite* tests the outcome of the method under test in several situations:

```
YearMonthWeekTest≫testDaysInMonth
  self assert:  (Month daysInMonth: 2 forYear: 2000) = 29.
  self assert:  (Month daysInMonth: 2 forYear: 2001) = 28.
  self assert:  (Month daysInMonth: 2 forYear: 2004) = 29.
  self assert:  (Month daysInMonth: 2 forYear: 2100) = 28.
```

3.2 One-method example commands

A *one-method example command* is a *one-method command* which does not have assertions for the method under test. So this command does not test the focused method against some desired result, but merely calls it. We detected three concrete instances of these commands:

3.2.1 Pessimistic one-method examples

A *pessimistic method example* is a *one-method example* which checks that an exception is thrown if a method is called in a way which violates a precondition. Beck [5] calls *pessimistic one-method examples "exception tests"*. Here is an example of a *pessimistic one-method example* ensuring that an attempt to create the directory C: on a Windows platform should fail:

```
DosFileDirectoryTests≫testFileDirectoryNonExistence
  "Hoping that you have 'C:' of course..."
  FileDirectory activeDirectoryClass == DosFileDirectory ifFalse:[^self].
  self
    should: [(FileDirectory basicNew fileOrDirectoryExists: 'C:')]
    raise: InvalidDirectoryError.
```

Note that we consider neither shouldnt: raise: nor should: raise: as assertions, because they do test whether something is true or false in a given state, but merely check whether or not an exception is thrown.

3.2.2 Optimistic method examples

An *optimistic method example* is a *one-method example* which expects that no exception is thrown if the method under test is called without violating some preconditions. Again, *optimistic method examples* do not contain *assertions*. The unit test below tests that the invocation of copyBits on a BitBlt in a certain situation does not throw an exception:

```
BitBLTClipBugs≫testDrawingWayOutside2
  | f1 bb f2 |
 f1 := Form extent: 100@100 depth: 1.
 f2 := Form extent: 100@100 depth: 1.
 bb := BitBlt toForm: f1.
 bb combinationRule: 3.
 bb sourceForm: f2.
 bb destOrigin: 0@0.
 bb width: SmallInteger maxVal squared; height: SmallInteger maxVal squared.
 self shouldnt:[bb copyBits] raise: Error.
```

3.2.3 One-method example suites

A *one-method example suite* is a *one-method example command* which calls the method under test more than once. It can be decomposed into several one-method command which call the same focused method once:

```
FractionTest≫testDegreeSin
  self shouldnt: [ (4/3) degreeSin] raise: Error.
  self assert: (1/3) degreeSin printString = '0.005817731354993834'
```

3.3 Multiple-method test suite

A *multiple-method test suite* is a *multiple-method command* which is decomposable into one-method tests. (See Figure 4).

3.3.1 Multi-facet test suites

Multi-facet test suites are *multiple-method test suites* that reuse a scenario to test several candidate methods. In the following example a previously initialized variable time is used to check different methods on Time.

```
TimeTest≫testPrinting
  self
    assert: time printString = '4:02:47 am';
    assert: time intervalString = '4 hours 2 minutes 47 seconds';
    assert: time print24 = '04:02:47';
    assert: time printMinutes = '4:02 am';
    assert: time hhmm24 = '0402'.
```

3.3.2 Cascaded test suites

Cascaded test suites are *multiple-scenario test suites* in which the results of one test are used to perform the next test:

```
Base64MimeConverterTest≫testMimeEncodeDecode
  | encoded |
 encoded ← Base64MimeConverter mimeEncode: message.
 self should: [encoded contents = 'SGkgVGhlcmUh'].
 self should:
   [(Base64MimeConverter mimeDecodeToChars: encoded) contents
     = message contents].
```

This *cascaded test suite* first triggers a method Base64MimeConverter≫mimeEncode:, tests its result encoded, and then uses encoded to test Base64MimeConverter≫mimeDecodeToChars:.

3.3.3 Independent test suite

An *independent test suite* is a *multiple-scenario test suite* which tests different methods on different receivers not depending on each other.

In the following example several independent methods are tested:

IslandVMTweaksTestCase≫replaceIn:from:to:with:startingAt: needs a totally different set of parameters than say

IslandVMTweaksTestCase≫nextInstanceAfter: [4]

```
IslandVMTweaksTestCase≫testForgivingPrims
| aPoint anotherPoint array1 array2 |
aPoint := Point x: 5 y: 6.
anotherPoint := Point x: 7 y: 8.  "make sure there are multiple points floating around"
anotherPoint.  "stop the compiler complaining about no uses"

self should: [ (self classOf:  aPoint) = Point ].
self should: [ (self instVarOf: aPoint at: 1) = 5 ].
self instVarOf: aPoint at: 2 put: 10.
self should: [ (self instVarOf: aPoint at: 2) = 10 ].

self someObject.
self nextObjectAfter: aPoint.

self should: [ (self someInstanceOf: Point) class = Point ].
self should: [ (self nextInstanceAfter: aPoint) class = Point ].


array1 := Array with: 1 with: 2 with: 3.
array2 := Array with: 4 with: 5 with: 6.

self replaceIn: array1 from: 2 to: 3 with: array2 startingAt: 1.
self should: [ array1 = #(1 4 5) ].
```

[4] Actually these tests are calling primitives, which are implemented in the virtual machine and not in the smalltalk image.

3.4 Others

We call all test cases which neither focus on one method nor are decomposable into one-method tests *others*.

3.4.1 Constraint test

A *constraint test* checks the interplay of several methods without focusing on one of them. In the following example a graphic conversion functionality is tested by comparing the original bitmap with the result obtained after encoding the bitmap to the png-format and then decoding it back again.

```
PNGReadWriterTest≫test16Bit
  self encodeAndDecodeForm: (self drawStuffOn: (Form extent: 33@33 depth: 16))
```

3.4.2 Meta test

A *meta test* is a test about the application itself, *e.g.,* its structure, its current state or its implemented or unimplemented methods. For example, the following test checks if the class of Metaclass only has one instance, namely Metaclass:

```
BCCMTest≫test07bmetaclassPointOfCircularity
  self assert: Metaclass class instanceCount = 1.
  self assert: Metaclass class someInstance == Metaclass.
```

3.4.3 Uncategorized

We call all unit tests which do not fall into one of the above categories *uncategorized*.

3.5 First validation: Maven

Using our taxonomy, we manually categorized 50 randomly selected JUnit tests of *Maven* [6], a Java project management and project comprehension [5].

25 of these tests merely checked some getter/setter code and were classified as constraint tests. The other sampled tests fell naturally into one of our proposed categories, and if less trivial getter/setter test code had been selected, we could expect again *one-method commands* as the majority of classified tests (See Figure 5).

[5] See http://www.iam.unibe.ch/∼gaelli/mavenUnitTests.html

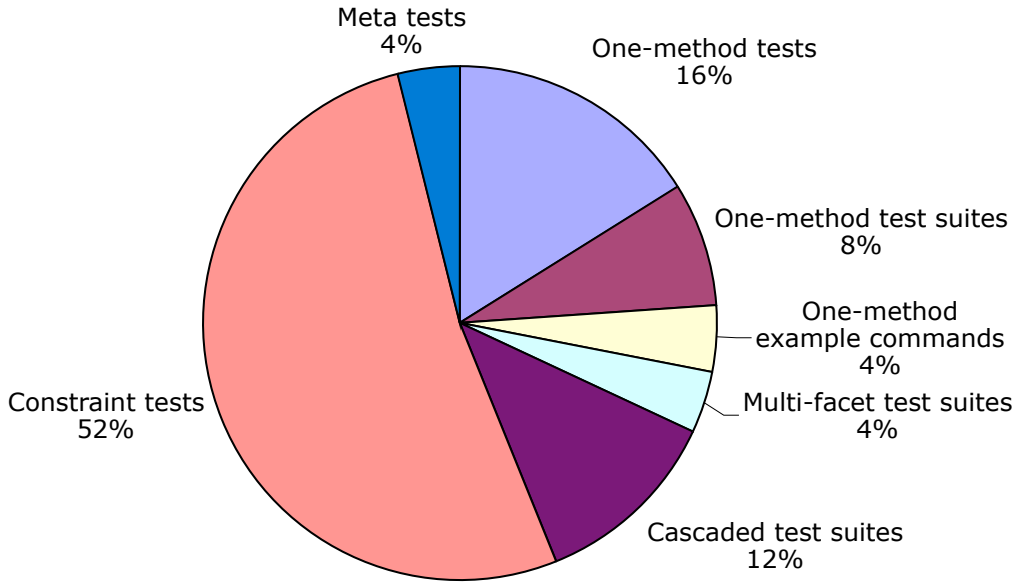

Fig. 5. Manual classification of 50 random unit tests of Maven

4 Automatic Classification of Unit Tests

After having manually derived the taxonomy, we developed some lightweight heuristics to automatically detect the feature properties depicted in Figure 2. Our goal is to classify most of the unit tests automatically. Using these heuristics we have been able to automatically classify 52% of the manually classified one-method commands tests, while our average precision rate was 89% (see Table 1). Finally we applied our automatic approach to a new case study and found that more than a third of the unit tests focus on single methods.

4.1 Instrumentation

To detect the feature properties we rely on dynamic analysis of the code, as we are dealing with runnable test cases in a dynamically typed environment.

Many of the unit tests of the Squeak base system test low level classes like Arrays *etc*. It is therefore not feasible to use method wrappers [7], because recursion would almost certainly arise when the wrapping algorithm uses a method which is about to be wrapped — thereby bringing our system to a halt. We therefore used the *bytecode interpreter* found in the class ContextPart, which is also used in the debugger of Squeak to step and send through methods.

Using and enhancing the bytecode interpreter of Squeak has the advantage of being more general than *method wrappers* and base level classes can be tested too. However, it comes with the following disadvantages:

- It is slower than current VM optimized method wrapper code.
- Simulation of exception handling code is buggy in the current implementation in the SqueakVM: As a consequence it did not work for exception handling code used by mainly by optimistic or pessimistic method examples.
- Methods which only return a variable are inlined by the Smalltalk-compiler and thus cannot be detected [6] .

4.2 Lightweight Heuristics

In the following we present a list of heuristics used to detect the feature properties displayed in the left subtree of the Figure 2. We have not yet developed any heuristics to classify leaves of the right subtree.

The first question in the decision tree is whether a unit test focuses on a single method. Three possible ways to detect this property are:

(1) *Deduction of the focused method from the command name*. One approach to deduce if a command focuses on one method is to examine the method name of the command. Often the developer includes the name of the method under test as part of the test method. A typical unit test looks like FooTest≫testBar which denotes that a method named bar of the class named Foo is tested and thus focused on. The execution of the test method can be simulated with our bytecode interpreter and thus checked, if it calls directly a method of the form Foo≫bar or Foo≫bar:.

If the naming convention of the test method name can be decoded and exactly one candidate method matches, then the developer has clearly indicated that this would be the method under focus. More specifically we deleted the first four characters "test" of the command name, and searched for a selector in the trace in the first level, that matches the remaining string, possibly converting the leading character to lower case, and ignoring parameters.

Example: If the test method name is BarTest≫testFoo then we look for an event in which a candidate method foo is called. If there are two selectors called, like foo: and foo, the result is ambiguous and we cannot say on which of them our test would focus.

(2) *Deduction of the focused method by the command structure*. We say that the command focuses on this method, if exactly one candidate method is called directly: A simple way to detect if a unit test focuses on one method is to find out if the test method only calls one candidate method, that is only one method of the package under test. This approach cannot be complete, as many unit tests do the setup of the test scenario not in the extra TestCase≫setUp method, but in the test method itself, and there they often have to call methods

[6] On the other hand this might be a welcome side effect as one would normally not focus a test on a method that merely returns a variable.

of the package under test for the setup. We do not make a distinction whether a candidate method is called only once or more than once, as long as it is the only called candidate method.

(3) *Deduction of the focused method by using historical information*. In incremental test-driven approaches the less complex methods will be built before the more complex ones. To test a more complex method the developer will likely refer to simpler candidate methods, either to build the scenario on which the complex method can be run or to use already existing methods as test oracles. However, in Squeak we do not know if a test case was developed before another test case, as Squeak still relies on a code exchange mechanism which destroys this versioning information.

To determine if a *one-method command* is a *one-method test command* or a *one-method example command* we check if it only calls `self should: [] raise: Exception`, `self shouldnt: [] raise: Exception` or friends, and if all the expressions inside the "shoulds" call the same method.

We can distinguish *one-method tests* from *one-method test suites* by simply counting how often the method under test is called. Accordingly we do the further split up in the right subtree, the *one-method example command* and then use the difference between the calls `should:raise:` and `shouldnt:raise:` to make the last distinction. With this heuristic we classify any *one-method test* as *one-method test command* which does not call any kind of `should:raise:` and `shouldnt:raise:`.

Category	**Manual result**	**Computed Result**	**Hits**	**Recall**	**Precision**
One-method tests	387	207	202	52%	98%
One-method test suites	114	86	57	50%	66%
Pessimistic method examples	11	15	10	91%	66%
Optimistic method examples	15	16	10	67%	63%
One-method example suites	10	1	1	10%	100%
Total	537	334	280	52%	89%

Table 1
Preliminary manual and automatic classifications of one-method commands of the Squeak Unit Tests.

4.3 A First Case Study: Squeak Unit Tests

Having categorized the Squeak Unit Tests before, we could compare the results of our lightweight heuristic with our manual results. (See Table 1). Squeak 3.7 has no notion of packages and relies on a naming convention of class-categories. We only

Category	Manual result	Computed Result	Hits	Recall	Precision
One-method tests	59	19	5	8%	26%
One-method test suites	80	48	37	46%	77%
One method example suites	3	3	3	100%	100%
Total	142	70	45	32%	64%

Table 2
Preliminary manual and automatic classifications of one-method commands of the SmallWiki Unit Tests.

automatically categorized 671 of 982 tests, whose class-category name allowed us to identify their package under test. Our heuristics were able to categorize 52% of the leaves of the left subtree from our taxonomy with a mean precision of 89%, meaning that only 11% of the categorized test cases were put in a different category than by the human reengineer.

4.4 A Second Case Study: SmallWiki

After having done a manual categorization (see Figure 6) we automatically categorized the 200 unit tests of *SmallWiki* [8], a collaborative content management tool written in VisualWorks Smalltalk and ported to Squeak. We chose this system as a case study, as it is a medium sized application developed by a single experienced developer in a test-driven way.

A surprising result here was that more tests could be detected as focusing on one method by considering the calls of only one candidate method, rather than by exploiting their naming convention.

We only programmed the detection for three categories, namely *one-method tests*, *one-method test suites*, and *one-method example suites*. All of them together represented already more than a third of all tests. Figure 6 shows that contrary to the Squeak case study, the developers here wrote more *one-method test suites* than *one-method tests*. The recall and precision for *one-method tests* displayed in Table 2 is only 5% respectively 26% as there have been many tests for getter/setter pairs: The getter-methods of variables are inlined and could thus not be detected by our bytecode interpreter. Only setter methods have been detected leading to false positives.

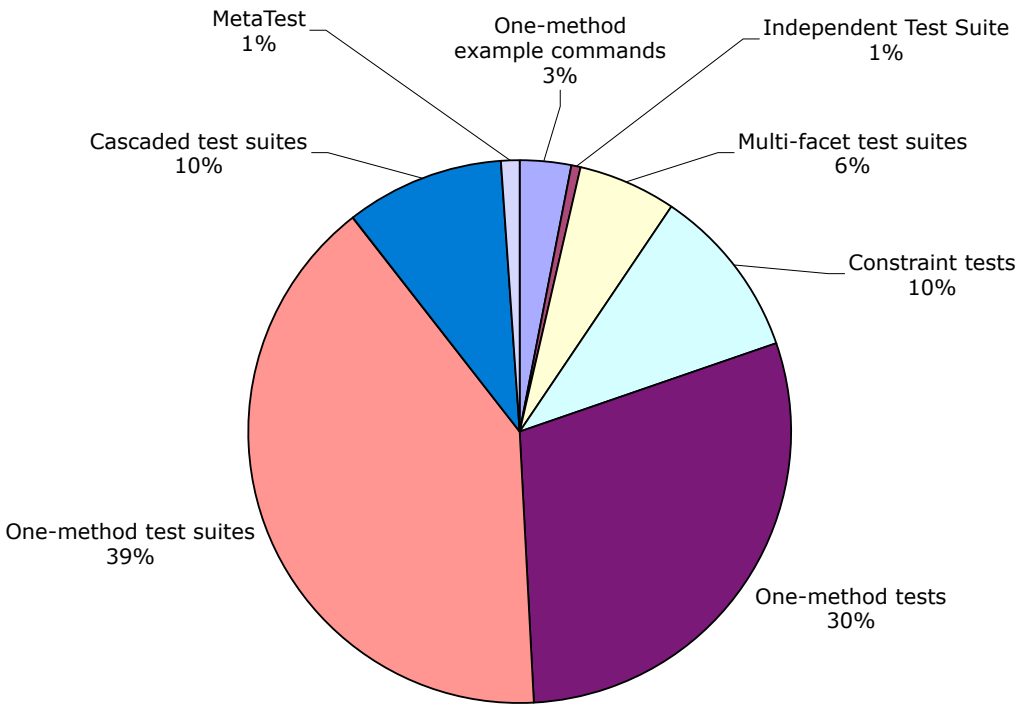

Fig. 6. Manual classification of unit tests for the SmallWiki system

5 Discussion

Although the taxonomy we have derived appears promising, it is a preliminary result for several reasons:

- Our taxonomy is based on only three case studies. Though it seldom arises that we discover new categories, more case studies need to be conducted.
- We focused on XUnit Tests, as described by Beck et al.[1] so we do not know if developers write other kinds of unit tests while using other testing frameworks.
- We have not addressed the question if unit tests should be considered whitebox or blackbox-tests and if they could likewise be used as acceptance, integration, or end-to-end tests.
- Only three of the Squeak Unit Test developers wrote 70% of the test cases making our sample data of this case study less representative.

Developers have complete freedom to write any kind of unit tests — making automatic classification a difficult business. The automatic classification heuristics are similarly preliminary and may fail in the following cases:

***.**Ambiguity of the naming convention Using the naming convention for automatic detection of the method under test is unreliable and ambiguous. For example, does the following test focus on Foo≫bar:, on Foo≫bar, or both of them? A similar problem arises in Java, as the naming convention will not differentiate between

overloaded methods that take different types of parameters.

```
FooTest≫testBar
  | aFoo |
  aFoo:= Foo new.
  aFoo bar: 1.
  self assert: (aFoo bar = 1)
```

We would manually categorize this one as a *constraint test*.

***.**Test framework tests Tests of the test framework may be incorrectly categorized. The following test could be classified as a pessimistic method example of error: but its intent is to be an *optimistic* method example of should:raise:

```
SUnitTest≫testException
  self
    should: [self error: 'foo']
    raise: TestResult error
```

***.**Assertions come only after clean up In some tests cleanups are necessary. As the cleanup does not have to influence the test result, developers also write the assertions after the cleanup.

In the following example both assertion statements could be moved two lines up preserving the test case. Thus it is activate and not wait or suspend which is tested.

```
StopwatchTest≫testMultipleTimings
 aStopwatch activate.
 aDelay wait.
 aStopwatch suspend.
 aStopwatch activate.
 aDelay wait.
 aStopwatch suspend.
 self assert: aStopwatch timespans size = 2.
 self assert:
  aStopwatch timespans first asDateAndTime <
  aStopwatch timespans last asDateAndTime
```

***.**Tested method is not the last called of the package under test Some tests are testing methods which are not the last method of the package called before the assertion occurred. Example: Is the method under test removeActionsWithReceiver: or actionForEvent:? The name of the command indicates the former, but the structure of the test suggests the latter:

```
EventManagerTest≫testRemoveActionsWithReceiver
 | action |
 eventSource
  when: #anEvent
  send: #size to: eventListener;
  when: #anEvent
  send: #getTrue to: self;
  when: #anEvent:
  send: #fizzbin to: self.
 eventSource removeActionsWithReceiver: self.
 action := eventSource actionForEvent: #anEvent.
 self assert: (action respondsTo: #receiver).
 self assert: ((action receiver == self) not)
```

***.**Mock objects The following test is interesting, as it is programmed by an experienced developer (it uses mock principles [9] to deal with program behavior). Here the methods under test in a cascaded scenario are overwritten so that additional information about the number of calls could be transcribed and tested. We currently subsume this kind of test under *meta tests*.

```
MorphTest≫testIntoWorldCollapseOutOfWorld
 | m1 m2 collapsed |
 "Create the guys"
 m1 := TestInWorldMorph new.
 m2 := TestInWorldMorph new.
 self assert: (m1 intoWorldCount = 0).
 self assert: (m1 outOfWorldCount = 0).
 self assert: (m2 intoWorldCount = 0).
 self assert: (m2 outOfWorldCount = 0).

 "add them to basic morph"
 morph addMorphFront: m1.
 m1 addMorphFront: m2.
 self assert: (m1 intoWorldCount = 0).
 self assert: (m1 outOfWorldCount = 0).
 self assert: (m2 intoWorldCount = 0).
 self assert: (m2 outOfWorldCount = 0).
  (...)
```

***.**Naming convention indicates one-method test, but it is not Which is the method under test here, weeks: or days? Days are computed too so it is also an interesting method to test. Our heuristic would detect Duration≫weeks as the method under test. We would manually categorize this one as a *constraint test*.

```
DurationTest≫testWeeks
 self assert: (Duration weeks: 1) days= 7.
```

***.**Developers do not agree on method under test Consider the two following tests written by two different developers: They both check if two different kinds of instantiations yield the same result. The name of the first indicates that it is testing =, the name of the second indicates that it tests the creation of instances. Both tests have at least two candidate methods, namely the instance creation methods and the = method.

```
IntervalTest≫testEquals4
  self assert: (3 to: 5 by: 2) = #(3 5).
  self deny: (3 to: 5 by: 2) = #(3 4 5).
  self deny: (3 to: 5 by: 2) = #().
  self assert: #(3 5) = (3 to: 5 by: 2).
  self deny: #(3 4 5) = (3 to: 5 by: 2).
  self deny: #() = (3 to: 5 by: 2).

MonthTest≫testInstanceCreation
  | m1 m2 |
  m1 := Month fromDate: '4 July 1998' asDate.
  m2 := Month month: #July year: 1998.
  self assert: month = m1.
  self assert: month = m2.
```

Any meaningful definition of *focuses on one method*, where at least two different candidate methods are involved, is likely to be dismissed by at least one of those

developers. As a compromise they could categorize both of them as *constraint tests*.

6 Related Work

Binder [10] discriminates between methods under test (*MUT*) and classes under test (*CUT*) but he does not discriminate between unit tests which focus on one or on several *MUTS*.

Beck [5] argues that isolated tests would lead to easier debugging and to systems with high cohesion and loose coupling. *One-method commands* are isolated tests, whereas *multiple method-commands* execute several tests and in the case of *cascaded method test suites* or *multi-facet test suites* depend on each other or on a common scenario.

Eclipse [11] provides a Search≫Referring Tests menu item which allows one to navigate from a method to a JUnit Test that executes this method. However no distinction is made between methods used for setting up the test scenario and those actually under test.

Jézéquel [12] discusses how testing can rely on the *Design by Contract principle* [13] and classes are seen as self-testable entities as much as possible by embedding unit test cases with the class. We found that developers write many tests we could categorize as *one-method commands*. The concept of *one-method commands* even makes methods self-testable. Squeak version 3.7 had almost 900 unit tests but only 24 assertions in the non test code. Associating *one-method examples* with assertion containing methods yields highly abstract *and* executable tests.

Van Deursen et al.[14] talk explicitly about unit tests that focus on one method and start to categorize them using bad smells like *indirect testing*, which describe tests that we would categorize as independent tests. In another paper [15] Van Deursen and Moonen explore the relationships between testing and refactoring, they suggest that refactoring of the code should be followed by refactoring of the tests. Many of these dependent test refactorings could be automated or at least made easier, if the exact relationships between the unit tests and their methods under test would be known.

Bruntink et al.[16] show that classes which depend on other classes require more test code and thus are more difficult to test than classes which are independent. Using *cascaded test suites*, where a test of a complex class can use the tests of its required classes to set up the complex test scenario, should improve the testability of complex classes.

Thomas [17] argues that the message-centric view deserves more attention. *One-*

method tests, *optimistic and pessimistic method examples* are all reifications of messages and are the atoms of all *one-method commands* and *multiple-method test suites*.

Edwards [18] is making a claim for *example centric programming*:

> In general, examples are standalone snippets of code that call the code under observation. Unit tests (...) are a good source of examples, and should be automatically recognized as such.

Our taxonomy should help us to link the different kinds of unit tests to the code they are exemplifying.

Test cases are implemented in XUnit using the "pluggable selector" pattern, which avoids the need to create a new class for each new test case at the cost of using the reflection capabilities of the system, thus making the *"code hard to analyze statically"* [5].

7 Conclusions and Future Work

We have developed a taxonomy which categorizes the relations

- between unit tests and methods under test and
- between unit tests and other unit tests.

Knowing these relations can help the developer to refactor, compose and run the program together with the tests, and thus to speed up their co-evolution. It can also help the reengineer to assess if a given method is adequately tested.

We have given initial evidence that the "unit" under test in object-oriented programs is most often a method and that most other kinds of unit tests can be decomposed into *one-method tests*.

We have started to develop some lightweight heuristics to automate this categorization. Our simple heuristics can identify a relevant portion of categories with a high precision rate. We have given evidence why complete automatic classification of unit tests using our taxonomy is impossible for all our suggested algorithms.

We have also discovered that developers write tests which do not have any assertion at all, but only establish whether a given method should or should not throw an exception: 5% of the tests in our manual case study and 2% in the automatic one fell into this category.

In the future we want to explore the following axes of research:

- We want to make the relationships between unit tests and methods under test explicit: First experiments show that if *one-method tests* also delivered the result of their *focused method* as a return value, one could parse the *one-method test* and clearly identify the *focused method*. This link also allowed the composition of tests, and would be stable to refactorings like renaming. Methods in statically typed languages can be void, thus we want to return a complex result object consisting of the receiver, parameters and possibly the return value of the *focused method*. We want to research the pros and cons of alternative denotations of the *focused method* using method comments, specific method sends or in case of Smalltalk bracketing blocks as markers.
- We want to evaluate if an optional 5-pane Smalltalk browser for navigating between tests and methods will be accepted by the Squeak community [19].
- We want to come up with heuristics to automatically categorize multiple method commands.
- We have previously proposed a partial order of unit tests by means of *coverage sets* — a unit test A *covers* a unit test B, if the set of method signatures invoked by A is a superset of the set of method signatures invoked by B [20]. In the four case studies we conducted, 75% of the unit tests were comparable to at least one other unit test in terms of that partial order. These results indicate that unit tests could be refactored into composed one-method tests leading to lower testing time and easier scenario building. We plan to enhance the IDEs of Squeak and Eclipse, so that developers can compose new tests from existing tests.
- We also plan to exploit this overlapping of many tests to identify focused methods under tests: If two tests TestA≫testOne and TestA≫testTwo directly call a method Foo≫foo but TestA≫testOne in addition calls only a method Bar≫bar, chances should be high, that TestA≫testOne is focusing on Bar≫bar.

We see this work as the beginning of the work on classifying unit tests and hope to spawn a discussion about this subject. For this reason we decided to put our taxonomy together with a nomenclature on our web site [7], so that we can easily integrate new kinds of unit tests we find or you report to us.

References

[1] K. Beck, E. Gamma, Test infected: Programmers love writing tests, Java Report 3 (7) (1998) 51–56.

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

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

[7] http://kilana.unibe.ch/nomenclatureofunittests/

[4] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, A practical Smalltalk written in itself, in: Proceedings OOPSLA '97, ACM Press, 1997, pp. 318–326.

[5] K. Beck, Test Driven Development: By Example, Addison-Wesley, 2003.

[6] Maven, http://maven.apache.org.

[7] J. Brant, B. Foote, R. Johnson, D. Roberts, Wrappers to the Rescue, in: Proceedings ECOOP '98, Vol. 1445 of LNCS, Springer-Verlag, 1998, pp. 396–417.

[8] L. Renggli, Smallwiki: Collaborative content management, Informatikprojekt, University of Bern (2003).

[9] T. Mackinnon, S. Freeman, P. Craig, Endotesting: Unit testing with mock objects (2000).

[10] R. V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Object Technology Series, Addison Wesley, 1999.

[11] Eclipse Platform: Technical Overview, http://www.eclipse.org/whitepapers/eclipse-overview.pdf (2003).

[12] J.-M. Jézéquel, Object-Oriented Software Engineering with Eiffel, Addison Wesley, 1996.

[13] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice-Hall, 1997.

[14] A. Deursen, L. Moonen, A. Bergh, G. Kok, Refactoring test code, in: M. Marchesi (Ed.), Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001), University of Cagliari, 2001, pp. 92–95.

[15] A. Deursen, L. Moonen, The video store revisited - thoughts on refactoring and testing, in: M. Marchesi, G. Succi (Eds.), Proceedings of the 3nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), 2002.

[16] M. Bruntink, A. van Deursen, Predicting class testability using object-oriented metrics, in: Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society Press, 2004.

[17] D. Thomas, Message oriented programming, Journal of Object Technology 3 (5) (2004) 7–12.

[18] J. Edwards, Example centric programming, in: OOPSLA 04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, ACM Press, 2004, pp. 124–124.

[19] M. Gälli, O. Nierstrasz, S. Ducasse, One-method commands: Linking methods and their tests, oOPSLA Workshop on Revival of Dynamic Languages (Oct. 2004).

[20] M. Gälli, M. Lanza, O. Nierstrasz, R. Wuyts, Ordering broken unit tests for focused debugging, in: 20th International Conference on Software Maintenance (ICSM 2004), 2004, pp. 114–123.

Co-evolving Code and Design with Intensional Views — A Case Study

Kim Mens *

Département d'Ingénierie Informatique (INGI)
Université catholique de Louvain (UCL)
Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium

Andy Kellens

Departement Informatica (DINF)
Vrije Universiteit Brussel (VUB)
Pleinlaan 2, B-1050 Brussel, Belgium

Frédéric Pluquet Roel Wuyts

Départment d'Informatique
Université Libre de Bruxelles (ULB)
Boulevard du Triomphe - CP212, B-1050 Bruxelles, Belgium

Abstract

Intensional views and relations have been proposed as a way of actively documenting high-level structural regularities in the source code of a software system. By checking conformance of these intensional views and relations against the source code, they supposedly facilitate a variety of software maintenance and evolution tasks. In this paper, by performing a case study on three different versions of the *SmallWiki* application, we critically analyze in how far the model of intensional views and its current generation of tools provide support for co-evolving high-level design and source code of a software system.

Key words: Case study, co-evolution, intensional views and relations, *SmallWiki*.

* Corresponding author.
Email addresses: Kim.Mens@info.ucl.ac.be (Kim Mens), akellens@vub.ac.be (Andy Kellens), fpluquet@yahoo.fr (Frédéric Pluquet), roel.wuyts@ulb.ac.be (Roel Wuyts).
URLs: http://www.info.ucl.ac.be/~km (Kim Mens), http://prog.vub.ac.be/~akellens/ (Andy Kellens), http://homepages.ulb.ac.be/~rowuyts/ (Roel Wuyts).

1 Introduction

Maintaining the source code of long-lived software systems requires an adequate documentation of their intended design. However, due to their constant evolution, it is often hard to keep their source code and design synchronized. This is partly due to the fact that current-day integrative development environments still focus too much on writing code and too little on supporting maintenance and evolution tasks [1].

Intensional source-code views and relations [2,3,4,5] have been proposed as an active documentation technique that addresses some of these problems. They increase our ability to understand and document the code and its design by grouping together structurally related source-code entities. They facilitate software maintenance and evolution, because alternative descriptions of the same intensional view can be checked for consistency and because relations between intensional views can be defined and verified against the source code.

In [2] we explained how to codify software architectures by means of intensional source-code views [1] and how to check conformance of those architectures with the source code. In [3] we proposed intensional views as an intuitive and lightweight but verifiable means of documenting crosscutting concerns in a software system. In [4] we discussed how intensional views facilitate a variety of software understanding, maintenance and evolution tasks. Finally, [5] emphasized on documenting and verifying high-level *relations* between intensional views. We also discussed the analogy of testing structural source-code regularities in a software system by means of intensional views and relations with testing the behavior of a software system by means of unit tests.

To define and verify intensional views and their relations we built a tool suite which we called *IntensiVE*. This 'Intensional View Environment' was implemented entirely in and seamlessly integrated with the *VisualWorks Smalltalk* development environment and comprises, amongst others, the following tools:

The Intensional View Editor (Fig. 1) allows us to document relevant concerns in the source code in terms of intensional views and to inspect the source-code entities corresponding to such concerns.

The View Consistency Checker (Fig. 2) allows us to verify consistency between different alternative descriptions of an intensional view, with respect to the current source-code base, and to provide fine-grained feedback on the differences between these alternative definitions.

The Relation Editor (Fig. 3) allows us to document high-level relationships between intensional views, as well as known deviations of these relationships in the source code.

[1] called 'virtual software classifications' in that paper

The Relation Checker (Fig. 4) allows us to verify these relations against the current source code, and provides fine-grained feedback on their validity.

Whereas older versions of these tools have been reported on briefly in [5], we have recently re-implemented them entirely to improve their efficiency, persistence and integration with version 2 of the *StarBrowser* [6], an advanced source code browser for *VisualWorks Smalltalk*. In addition to having the logic query language *Soul* [7] as underlying language in which to describe the intensional views and relations, the tools now offer support for using *Smalltalk* too as query language to reason about source code. Another novel feature is the ability to define nested views, which allows us to create context-specific views. Finally and most importantly, we added support for visualizing intensional views and relations (see Fig. 5), by relying on *CodeCrawler* [8], a reverse engineering tool which combines software metrics and visualization.

The aim of this paper is to perform a critical evaluation of the current generation of tools, including the new opportunities offered by the visualization tool, to support co-evolution of high-level design and source-code of a medium-sized *Smalltalk* application. The case we selected for this study is *SmallWiki* [9], an object-oriented Wiki implementation in *Smalltalk*. We documented the intended design of an early version of *SmallWiki* and observed how this documentation helped us in better understanding the software and its implementation structure, as well as in discovering certain structural irregularities in its source code. Then we verified this design documentation against two more recent versions of *SmallWiki* and discovered some interesting ways in which the source code and its design evolved.

From the experiences gained with this case study, we distilled a list of lessons learned about the model of intensional views and relations and its associated tools, in particular on how they support co-evolution of source code and higher-level design. Amongst others we learned that documenting the design of a software system with intensional views and relations allowed us not only to detect interesting structural inconsistencies introduced in the code upon evolution, but also that the process of documenting itself helped us to better understand the source code and how it evolved. A dedicated visualization which highlights what views and relations have become inconsistent with the code, proved very useful since it allowed us to readily assess the impact of an evolution step and locate potential structural problems. Finally, the ability of using and combining both logic and *Smalltalk* queries had the advantage that we could always choose the query language most appropriate to our needs, that is, the one that yields the most compact and declarative queries.

2 Experimental Setup: SmallWiki

A Wiki is a collaborative web application that allows users to add content, but also allows anyone to edit the content. *SmallWiki* [9] is a fully object-oriented and extensible Wiki framework that was developed entirely in *VisualWorks Smalltalk*. As opposed to most other Wiki implementations, which are hard to adapt, SmallWiki has been designed from the start with extensibility in mind. It has a clean object-oriented design where all entities that can be stored in web pages (text, links, tables, lists) are explicitly modelled as objects. Everything in *SmallWiki* is designed to be extended: page types, storage mechanism, actions, security mechanism, web-server, etc. Plug-ins can be shared within the community and loaded independently of each other into the system.

We decided to use *SmallWiki* for our case study for several reasons. Because it is open source, its source code is freely available. Secondly, many versions exist, from very early versions up until the stable versions that are currently in use at several places. Thirdly, it is a non-trivial piece of software, yet still manageable in size and complexity. We studied the following versions of *SmallWiki*:

Version 1.54 (14-12-2002) was the first internal release of *SmallWiki*, offering an operational Wiki server with rather limited functionality: only the rendering and editing of fairly simple Wiki pages was supported. This version contained 63 classes and 424 methods.

Version 1.90 (15-01-2003) covered only one extra month of development (thus limiting the risk of having a version that was too drastically different from the first version studied). Nevertheless, this month represented quite an active period of development with several releases a day (thus making it a non-trivial version to study). This version contained 8 more classes (71 in total) but many more methods (633). An important change with respect to version 1.54 was that in this newer version the methods responsible for rendering HTML code were refactored.

Version 1.304 (16-11-2003) was chosen because it covered a larger development period (almost 1 year) with lots of intermediate versions. This allowed us to study the problem of synchronizing design documentation and source code over a longer time interval. With 108 classes and 1219 methods, this version was significantly larger than the previous two.

In order to study the usefulness of intensional views and relations to document the design structure of an evolving software system, we conducted the following experiments on the different versions:

(1) We started by codifying the design of version 1.54 and investigated how this documentation helped us in better understanding the code structure as well as some of the adopted naming and coding conventions.

(2) We then verified this structural documentation against the more recent version 1.90 and drew conclusions about how SmallWiki evolved, and about the consequences of this evolution on the documented structure.
(3) Finally, we verified the documentation against the most recent version studied (1.304) and observed that the design remained relatively stable, even after this longer development period.

3 *IntensiVE*

Before describing our experiments in more detail, in this section we give an overview of the model of Intensional Views and Relations, together with its associated tool suite: *IntensiVE*, or Intensional View Environment. The following five subsections each focus on one of the major sub-tools of the environment namely the intensional view editor, the view consistency checker, the relation editor, the relation checker, and the intensional view displayer. Along the way we explain the underlying model of intensional views and relations.

3.1 The Intensional View Editor

An *Intensional View* is a set of source-code entities (classes or methods) which are structurally similar. Instead of enumerating all elements that make up a view, it is defined by means of an *intension*: an executable description which yields, upon execution, the set of entities belonging to the view, also called the *extension* of the view.

The *Intensional View Editor* (Fig. 1) is our main tool for creating and manipulating views. On the screenshot, the left pane shows all defined views in a tree representation. The right hand side shows the Intensional View Editor opened on a view named 'Execute Methods'. This view groups all methods responsible for executing actions on Wiki pages. Since all these methods are classified in an 'action' method protocol, we provide the following intension for the Execute Methods view: `methodInProtocol(?entity,action)`. This query, written in the logic language *Soul* [7], binds occurrences of methods in the 'action' protocol to the free logic variable `?entity`. By convention, the Intensional View Editor assumes that a logic query has a free variable named `?entity` and calculates the view extension as the accumulation of all bindings to that variable. When using *Smalltalk* as query language, it suffices to write a *Smalltalk* block that returns a collection. E.g., we can define a view of all *SmallWiki* classes by means of a *Smalltalk* expression `SmallWiki allClasses`.

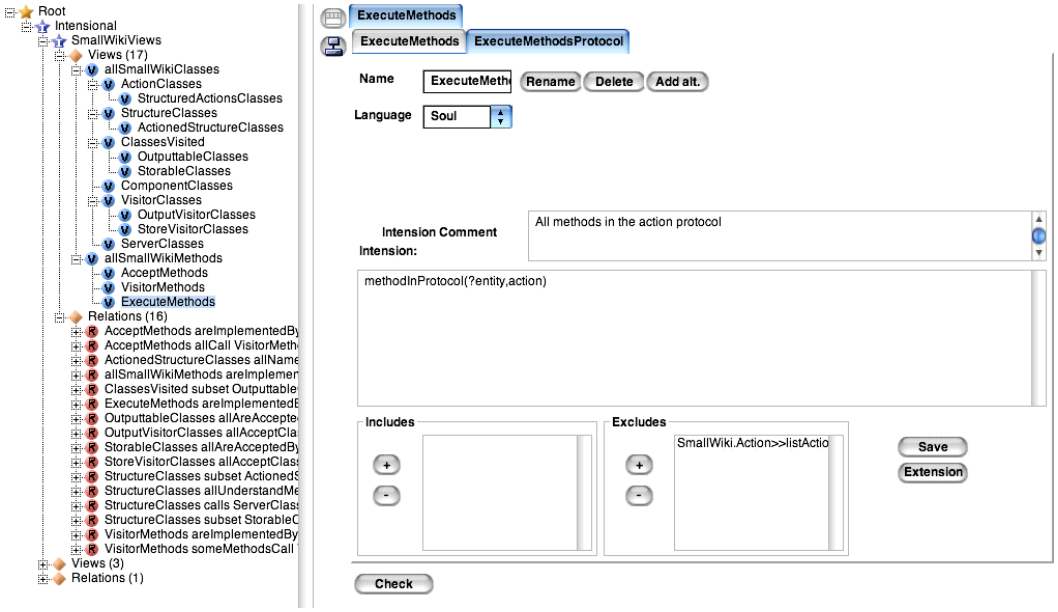

Fig. 1. The Intensional View Editor at work

Notice in the screenshot (left pane) that this view is defined as a subview of the view containing 'all SmallWiki methods'. The semantics of defining a view as subview of another one is that the intension of the subview is calculated in the context of the parent view. In other words, evaluating the intension of the Execute Methods view results in all methods which belong to the extension of the view 'all SmallWiki methods' but also to an `action` method protocol.

The tool also supports the explicit exclusion (resp. inclusion) of an entity from a view. For example, the method `listActions`, implemented on the `Action` class, is part of the computed extension of the Execute Methods view, but is not really an execute method. Hence we explicitly excluded it from the view, by putting it in the 'excludes set' of the view. Analogously, we have an 'includes set' of entities that should be included in a view, even though they do not satisfy the intension.

Intensional Views allow the definition of multiple alternative descriptions for the same view. This ability, together with the requirement of extensionally consistency (explained in the next subsection), provides an elegant way of declaring interesting naming and coding conventions to be respected by the entities of a view, as we will see in Section 4.

3.2 The View Consistency Checker

Fig. 2 shows the *View Consistency Checker*. This tool is used to verify that the different alternative descriptions of a same view are *extensionally consistent*, meaning that they all produce the same extension. When this constraint is violated, the tool

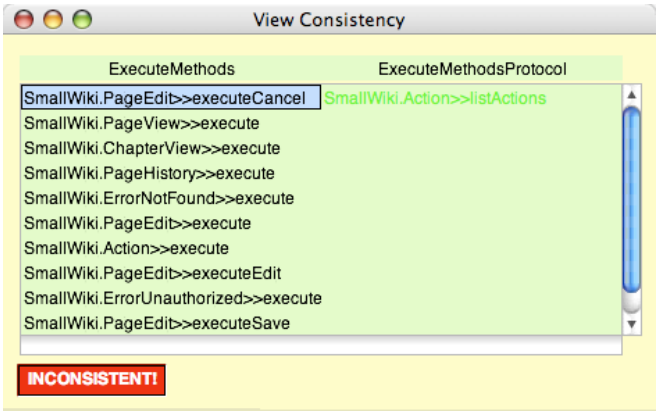

Fig. 2. The View Consistency Checker at work

provides appropriate feedback on what entities are in cause.

To illustrate this consider the Execute Methods view again. In addition to the intension already described above, we defined an alternative description based on the observation that the names of all execute methods start with the string 'execute'. Fig. 2 shows the result of checking extensional consistency between these two alternatives of the Execute Methods view. Note that we checked extensional consistency before having explicitly excluded `listActions` from the second alternative of the view. In fact, it was precisely the feedback from the View Consistency Checker that motivated us to take a look at the implementation of that method and decide that it was a deviating case.

The tool shows the user a column per alternative description of the view. The first column contains the extension of the main alternative (by default this is the first alternative of the view, but double-clicking a column changes the main alternative); the other columns contain the delta between the extension of the main alternative and the alternative represented by the column. If an element does not exist in the main alternative, it is coloured green. Elements present in the main alternative, but not in the other are displayed in red.

3.3 The Relation Editor

The *Relation Editor* allows a user to document relations between intensional views. Our model currently supports only relations of the canonical form:

$$\mathcal{Q}_1\ x \in Source : \mathcal{Q}_2\ y \in Target : x\ R\ y$$

where $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are either logic quantifiers $\forall$, $\exists$, $\exists!$, $\nexists$ or more fuzzy quantifiers [2]

[2] The fuzzy quantifiers are defined in terms of a minimum or maximum number of elements for which the condition should hold.

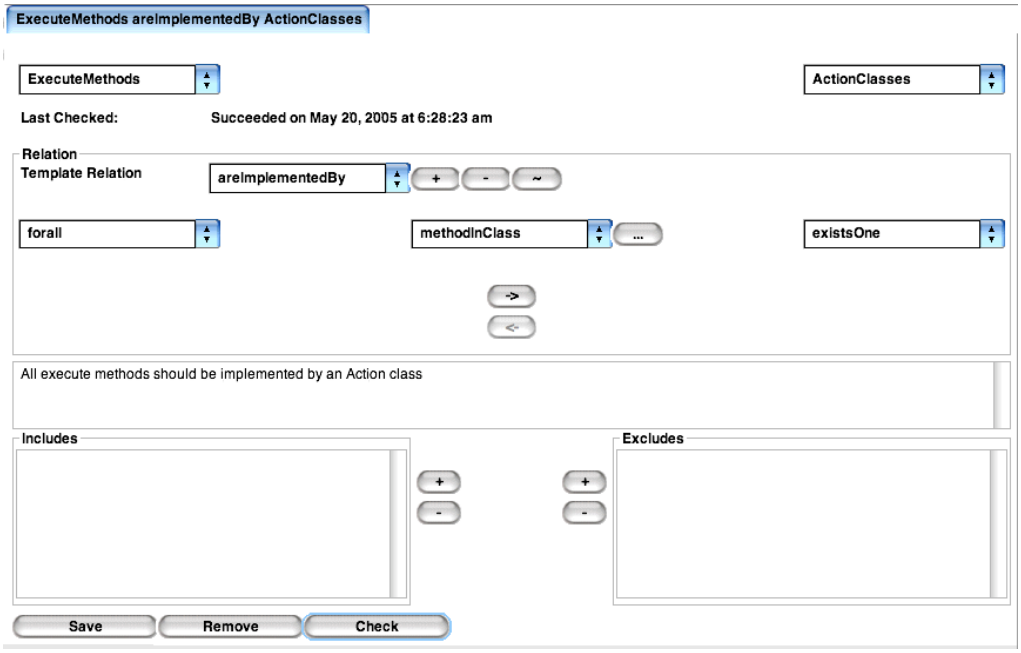

Fig. 3. The Relation Editor at work

like $some$, few, $many$ or $most$. $Source$ and $Target$ represent intensional views and R is a binary predicate over the source-code entities (denoted by x and y) contained in those views. A simple example of an intensional relation is that all Execute Methods are implemented by an Action Class (we define this view in Section 4.1). Fig. 3 shows the Relation Editor opened on this relation. Expressed in the canonical form above, the relation was defined as:

$$\forall\, x \in \textit{ExecuteMethods} : \exists!\, y \in \textit{ActionClasses} : x\ methodInClass\ y$$

To define a binary predicate R over source-code entities, in terms of which intensional relations can be defined, our tool offers two possibilities. In addition to defining the predicate directly in *Smalltalk* (using a *Smalltalk* block that takes two arguments and returns a boolean), the user can opt to use a *Soul* predicate (typically using LiCoR, an extensive library of *Soul* predicates to reason about source code). For concrete examples we refer to Subsection 4.2.

Like the Intensional View Editor, the Relation Editor supports the explicit declaration of deviating cases. It allows a user to specify explicitly tuples of source-code entities to be included in or excluded from the relation.

3.4 The Relation Checker

When pressing the 'Check' button in the Relation Editor (Fig. 3), the validity of a relation with respect to the source code is checked and the user is presented an instance of the *Relation Checker* (Fig. 4). Besides reporting whether the relation

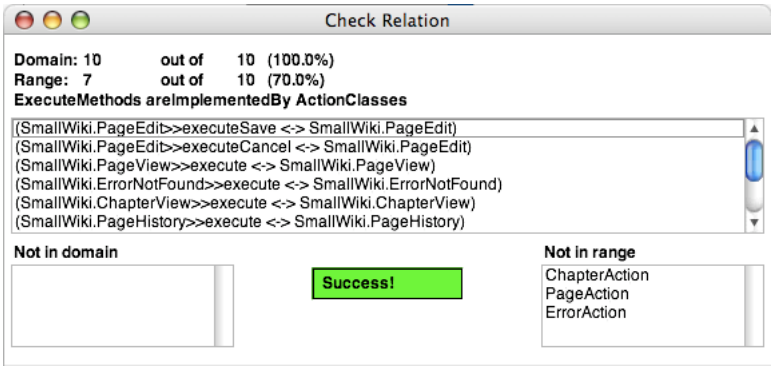

Fig. 4. The Relation Checker at work

holds, the tool presents the user a list of all tuples for which the relation is valid as well as some statistics on how many elements from source and target participate in the relation. It also lists all entities from the source view which are *not in the domain* of the relation as well as all entities in the target which are not reached by the relation. When a relation does not succeed, a user can use this information to determine for which source code entities the documented relation and the source code are no longer synchronized.

3.5 The Intensional View Displayer

All tools above support a user in manipulating (declaring, modifying, renaming, removing, verifying and saving) intensional views and relations. What is still missing is a visualization tool that provides a user with a global and compact drawing of all defined views and relations (or a relevant subset thereof). This is the purpose of the *Intensional View Displayer* depicted in Fig. 5. For a given selection of views, the displayer shows all these views, all their alternative descriptions, all subview links and all intensional relations in which those views take part. The views are laid out automatically in a hierarchy that reflects the view nesting, but the layout can be modified and stored manually.

Since the visualization tool is defined on top of *CodeCrawler* [8], a reverse engineering tool which combines software metrics and visualization, by making intelligent use of metrics we can highlight important characteristics of intensional views or relations. For example, a simple metric for a view is the number of entities contained in its extension. In Fig. 5 this metric was used as height of the rectangular boxes representing the views. For example, we can see that the view All SmallWiki Classes has many more entities than the Action Classes view, which is normal because the latter is defined as a subview of the former.

The visualization tool also uses colors to distinguish the different kinds of objects in a drawing. By default, the name and rectangle of intensional views are drawn in black, as well as the subview edges (starting with a triangle) and edges relay-

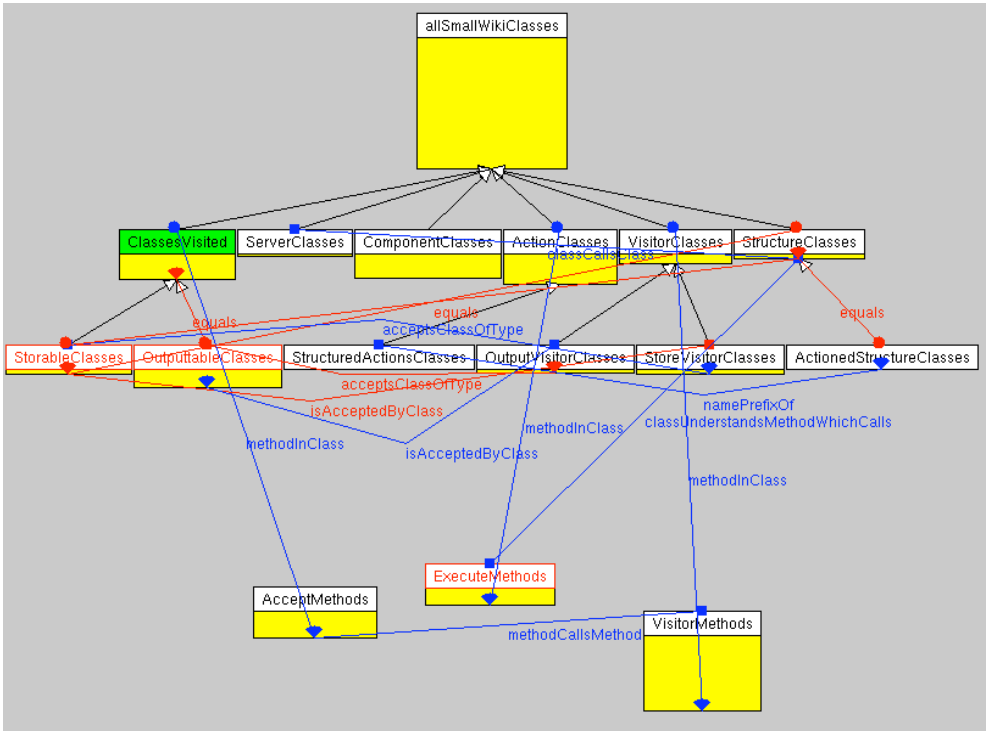

Fig. 5. The Intensional View Displayer at work on *SmallWiki* 1.304

ing a view with its alternative descriptions (ending with a diamond). The text and rectangle of the alternative descriptions are rendered in grey and an option can be toggled to not render them at all. Finally, edges representing intensional relations, together with the relation name, are drawn in blue. What is more interesting is that colors can be used as a metric too, for example to highlight inconsistencies in the documentation. The 'View Consistency' metric, for example, calculates the extensional consistency of a view and draws the view in red when inconsistent. A similar metric can be applied to the links connecting a view to its alternatives, to indicate what particular alternatives are inconsistent. In a similar way a color metric can be applied to the intensional relations, so that invalid intensional relations are highlighted in red. E.g., Fig. 5 immediately tells us that the view *Outputtable Classes* is inconsistent with some of its alternatives, and that the relation between *Classes Visited* and *Outputtable Classes* is invalid too: they are all colored red.

4 Experiment 1 (Documenting the structure of *SmallWiki* 1.54)

Having explained the *IntensiVE* toolsuite in detail, we now elaborate on the actual experiments we conducted on *SmallWiki*. In our first experiment we tried to document the intended design of *SmallWiki* version 1.54 and investigated how this documentation helped us in better understanding the implementation structure as well as some of the naming and coding conventions that were used. Due to a lack

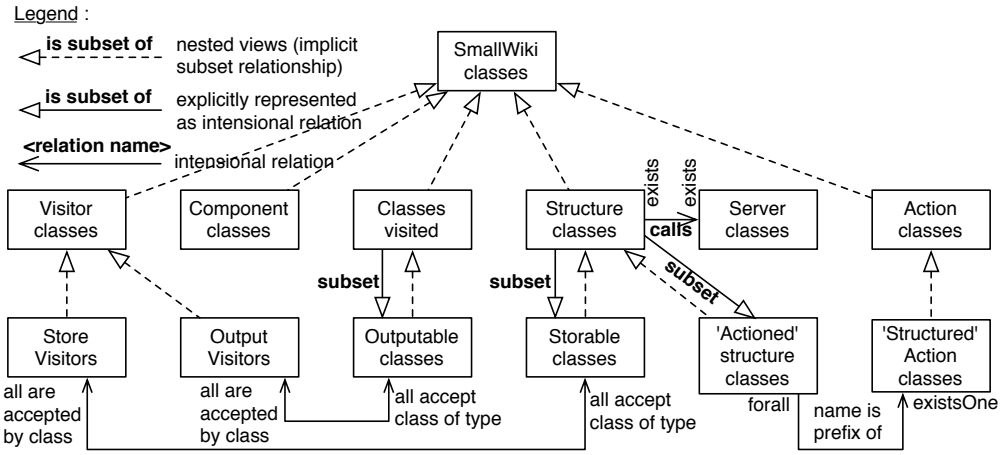

Fig. 6. Intensional Views and Relations on SmallWiki

of adequate documentation for this particular version, the approach we adopted was largely manual. We manually inspected the code, looking for interesting groups of classes or methods, codified those groups as intensional views, checked the views against the source code and further refined them when necessary, inspected the elements of the defined views to uncover relations with other views (potentially to be defined), etc.

In total, we came up with 17 intensional views, related by 14 nesting relationships and 16 intensional relations. Figures 6 and 7 summarize all defined views and relations. Whereas Fig. 6 shows the views containing classes and their interrelationships, Fig. 7 focusses on views containing methods. Next two subsections first discuss the views and then the relations between the views.

4.1 Views

All SmallWiki Classes. First of all we defined a view consisting of all classes in the application under study. This view was codified straightforwardly by means of a *Smalltalk* query `SmallWiki allClasses`.[3]

To restrict their domain to the *SmallWiki* classes only, the rest of the views were defined as subviews of this view. For example, we defined a series of views corresponding to the important class hierarchies in the code. They were all defined by means of a *Soul* query of the form
`classInHierarchyOf(?entity,[`*root class of hierarchy*`])`.

Structure Classes (classes in hierarchy of `Structure`) represent *SmallWiki* entities that can be referred to by a single URL, like a web page.

[3] Or alternatively using a *Soul* query `classInNamespace(?entity,[SmallWiki])`.

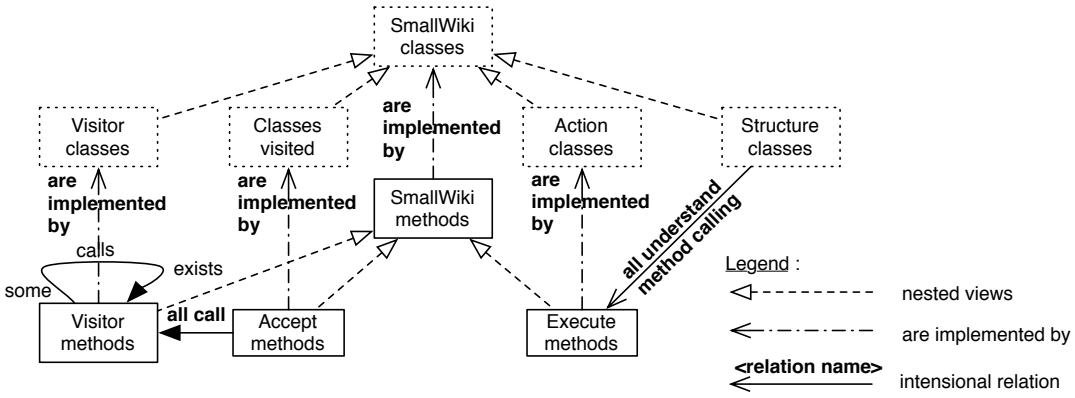

Fig. 7. Relations on SmallWiki method views

Component Classes (`PageComponent` hierarchy) represent the components out of which a web page can be constructed: text, links, tables, lists, ...

Visitor Classes (`Visitor` hierarchy) visit the structure and component classes and play a crucial role in *SmallWiki*, e.g. for rendering and storing web pages.

Action Classes (`Action` hierarchy) model the actions that can be performed on Wiki pages.

Server Classes (`WikiServer` hierarchy) represent the different kind of Wiki servers supported by *SmallWiki* (version 1.54 only supported Swazoo).

Other views that we defined, mainly by manual code inspection, were:

Visitor methods are all methods implemented in the `Visitor` class hierarchy that belong to a visiting method protocol. In *Soul* this was expressed as:

```
classInHierarchyOf(?class,[SmallWiki.Visitor]),
methodOfClassInProtocol(?entity,?class,?protocol),
['visiting*' match:?protocol]
```

In fact this is an example of a hybrid query where we use logic to reason about the code structure and evaluate a *Smalltalk* expression, parameterized by a logic variable (this is a particular feature of *Soul*), to reason about strings.

Accept methods are the methods named `accept:` and play an important role in the Visitor design pattern. We defined this view by means of a *Soul* query `methodWithName(?entity,[#accept:`

Actioned structure classes We defined the group of all structure classes on which actions can be performed as a subview of the Structure Classes. They can be recognized easily because they have a corresponding `Action` class:

```
classInViewNamed(?c, ActionClasses),
['*Action' match: ?c name],
[(?entity name, 'Action') = ?c name asString]
```

Structured action classes Dually we defined the view of action classes for a particular structure class as a subview of the Action Classes.

Execute methods are responsible for executing different actions on Wiki pages, such as rendering, saving, canceling and editing. They have in common that their names start with 'execute'. We defined this view by means of the intension: `['execute*' match: ?entity selector asString]`. Because we observed that the *SmallWiki* developer(s) consistently adopted the convention to put these methods in an 'action' method protocol, we also defined an alternative intension: `methodInProtocol(?entity,action)`.

Defining the views above triggered the definition of some more views:

Store Visitors and Output Visitors After having defined the Classes Visited view we wondered what classes were being visited and for what reason. By inspecting the Visitor Classes in more detail we learned that in *SmallWiki* 1.54 there were two main visitors: a 'store' visitor and an 'output' visitor. We codified these straightforwardly as subviews of the Visitor Classes view: all Visitor classes named `VisitorStore*` or `VisitorOutput*`, respectively.

Storable Classes and Outputtable Classes We also defined a view representing the 'storable' classes, i.e. classes visited by a Store Visitor, and one representing the 'outputtable' classes, as subviews of Classes Visited. We only show the definition of the Storable Classes, the one for Outputtable Classes being analogous. We defined the view in terms of the newly defined Store Visitors: the store visitor classes need to implement a specific method `accept:` for every class they want to visit. Without divulging all details, the following hybrid query extracts these visited classes from the names of the Store Visitors:

```
classInViewNamed(?class,StoreVisitors),
methodWithNameInClass(?method,?selector,?class),
[?selector = (#accept, ?entity name, ':') asSymbol]
```

As a second example of how existing views were reused to gradually refine and understand the code structure, the definition of the Visitor Classes view triggered the definition of a view consisting of all classes *being* visited:

Classes Visited are those classes that can be visited by Visitor Classes. Since the Visitor design pattern [10] uses a double dispatch protocol where the visited classes implement an accept method taking a visitor as argument, we defined this view using the *Soul* query `methodWithNameInClass(?M,[#accept:],?entity)`. In addition, since all these `accept:` methods belonged to a 'visiting' method protocol, as alternative description we used `protocolInClass(visiting,?entity)`.

When checking consistency of this view, we learned that the use of the 'visiting' protocol was indeed adhered to in a very disciplined way: all classes implementing `accept:` also

had a 'visiting' method protocol and vice versa. To document this, we defined the following alternative for the previously discussed *Accept Methods* view: `methodInProtocol(?entity,visiting)`.

However, since all alternative descriptions should produce the same extension, this implied not only that every `accept:` method belongs to a `visiting` method protocol but also that every method in a `visiting` method protocol is an `accept:` method. That constraint was clearly too strong, as we learned when verifying it using the View Consistency Checker: the `Visitor` class did not implement an `accept:` method, but did contain a few 'visit' methods in the visiting protocol. By excluding the `Visitor` class from the new alternative, the constraint became valid.

We noticed that there are quite some views (and relations, as we will see in Subsection 4.2) in our design documentation that document the Visitor design pattern [10], even though that specific pattern is quite well known and well understood. Nevertheless, we decided to document it explicitly because of the crucial role the pattern plays in the *SmallWiki* implementation, but more importantly because we wanted to be able to verify whether the implementation constraints implied by this pattern remained consistently adhered to in future versions of *SmallWiki*.

4.2 Relations between intensional views

All relations we identified between intensional views containing classes are summarized in Fig. 6. Dashed lines ending with a triangle represent view nesting. In addition to those subset relationships we codified some extra subset relationships between non-nested views:

Classes Visited *is subset of* Outputtable Classes Not only are all outputable classes a particular kind of visited classes (which was codified by means of nesting), in fact *all* visited classes are outputtable.

Structure Classes *is subset of* Storable Classes Whereas all visited classes are outputtable, only a few are storable. On the other hand, all structure classes, with the notable exception of the abstract superclass `Structure` itself, were storable. Since this seemed like a potentially important design contstraint, we documented it as an intensional relation with an explicit deviation for the exceptional case of the `Structure` class.

Structure Classes *is subset of* Actioned Structure Classes Although the 'actioned' structure classes were defined as subview of the structure classes, we observed that all structure classes (again with the exception of the class `Structure`) were 'actioned', i.e. had a corresponding `*Action` class.

'Actioned' Structure Classes versus 'Structured' Action Classes This same observation led us to define the following intensional relation between the 'Actioned' Structure Classes and 'Structured' Action Classes :

$\forall\ x \in$ *ActionedStructureClasses* : $\exists!\ y \in$ *StructuredActionClasses* :
$x\ has\ name\ which\ is\ prefix\ of\ name\ of\ y$

where the relation predicate was defined using the following *Smalltalk* block:

```
[:class1 :class2 | (class1 name asString),'*'
                   match: (class2 name asString)]
```

Next we defined the relationship between the visitors and the visited classes.

Output Visitors *all accept class of type* Outputtable Classes and
Store Visitors *all accept class of type* Storable Classes
Because of the double dispatch mechanism used in the visitor design pattern we know that *all* visitor classes that can handle a certain type of class need to implement *a* specific accept method taking objects of that type as argument. In particular this holds for the output visitors and outputtable classes, as well as for the store visitors and storable classes. The (hybrid) *Soul* predicate in terms of which we defined these intensional relations is given below. Due to the lack of static typing in *Smalltalk*, the predicate relies on the fact that the formal parameters of the method are named after the expected type.

```
acceptsClassOfType(?VisitorClass,?VisitedClass) if
   methodWithNameInClass(?Method,?Selector,?VisitorClass),
   ['accept*' match:?Selector asString],
   argumentOfMethod(?Argument,?Method),
   ['*',(?VisitedClass name asString),'*' match:?Argument asString]
```

Outputtable Classes *all are accepted by* Output Visitors and
Storable Classes *all are accepted by* Store Visitors
Conversely, all visited classes are supposed to be accepted by at least one visitor class. In particular this holds for the outputtable classes and output visitors, as well as for the storable classes and store visitors. The logic predicate in terms of which this relation is defined, is the inverse of the above:

```
isAcceptedByClass(?VisitedClass,?VisitorClass) if
   acceptsClassOfType(?VisitorClass,?VisitedClass)
```

We also documented that server classes are invoked by structure classes.

Structure Classes calls Server Classes Since not all server classes need to be invoked (it suffices to have one server running) and not all structure classes call the server classes, this intensional relation was defined as
$\exists\, x \in$ *StructureClasses* : $\exists\, y \in$ *ServerClasses* : $x\ classCallsClass\ y$
where $x\ classCallsClass\ y$ checks if class x has a method that potentially calls a method on class y. This predicate was taken from the logic library.

Whereas Fig. 6 focused on views containing classes, Fig. 7 summarizes the relations between intensional views containing methods. First of all there are the obvious implementation relationships:

Accept Methods *are Implemented By* Classes Visited
Execute Methods *are Implemented By* Action Classes
Visitor Methods *are Implemented By* Visitor Classes

Other intensional relations which we documented were:

Accept Methods *all call* Visitor Methods Indeed, the `accept:` methods all have the following pattern to call an appropriate method on the visitor:

```
accept: aVisitor
   aVisitor accept<name of class>: self
```

To express this relation we used a universal quantifier and a predicate `methodCallsMethod` which we declared in *Smalltalk* using the following block:

```
[:method1 :method2 | method1 sendsSelector:
                     (method2 compiledMethod selector)]
```

Visitor Methods *some call* Visitor Methods This relation codifies the fact that a visitor method is often implemented in terms of other visitor methods. For example, quite some of the `accept*` visitor methods make a self call to the `visit:` method. For expressing this intensional relation we used the same predicate as above and a fuzzy quantifier $some$ which requires that the relation is valid for at least 25% of the elements in its domain.

Structure Classes *all understand method calling* Execute Methods Since actions are to be executed on things like web pages, which are represented by structure classes, we require that all these structure classes understand (at least one) method that calls an appropriate execute method for actually handling the actions. For example, the abstract class `Structure` implements a method named `evaluateActionWithRequest:response:` which calls `execute` on the appropriate action class. We defined this in terms of a logic predicate which we added to the logic library.

5 Experiment 2 (Comparing the documentation with *SmallWiki* 1.90)

In the second experiment we compared the documented design of version 1.54 to the more recent version 1.90 and tried to understand how *SmallWiki* evolved, and what the consequences of this evolution were on the design documentation. To do so, we loaded the new version and recomputed and visualized all known intensional views and relations with the Intensional View Displayer. As explained in Subsection 3.5 and illustrated in Fig. 8, all conflicting views and relations were highlighted in red. We inspected the conflicts and tried to understand the discovered problems.

Inconsistent views

Storable Classes became inconsistent because of an explicit deviation in the third alternative that was no longer needed. More precisely, we originally documented that all Storable Classes except the `Document` class were Structure Classes. In version 1.90, the system had been refactored such that the `Document` class was moved to another hierarchy. We updated the design documentation by removing the deviation.

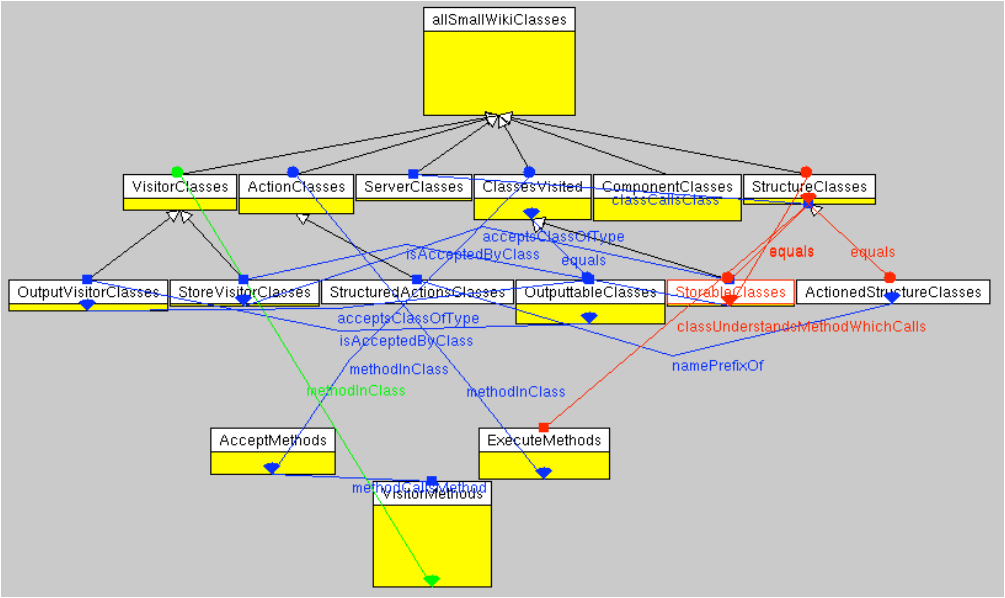

Fig. 8. Intensional views and relations on version 1.90

Invalid relations

Structure Classes *all understand method calling* Execute Methods
failed because in *SmallWiki* 1.54 the execute methods were being abused to render web pages in HTML format. In the newer and cleaner version, page rendering was performed by separate rendering methods. The only remaining purpose of the `execute` methods was to dispatch action requests to more appropriate methods depending on the action to be taken. Hence the invalid relation highlighted an interesting restructuring of the application. To update the documentation we did two things:

(1) we documented the dispatching mechanism by defining an intensional relation which required the `execute` methods to make a self call to a more specific `execute*` method (where `*` is a non-empty string). E.g., the method `execute` on class `ChapterEdit` calls either `executeCancel` or `executeSave` if the corresponding button was selected and performs an `executeEdit` otherwise;

(2) we documented that the `execute` methods were not allowed to send messages to the instance variable named `html` (which was typically the way how rendering was being done).

After having done so we still found a few violations against these new constraints but did not codify them as explicit deviations, since we wanted to emphasize that they were real design conflicts that should be fixed in a new version of the code.

Structure Classes *is subset of* Storable Classes failed because of the addition of two new intermediate superclasses. We defined these two classes as deviating cases of the relation.

Comparing view sizes

Using the Intensional View Displayer, we compared the size of all views on version 1.54 with those on version 1.90. We wanted to find out if and where there were important dif-

ferences in size, as these may indicate potential problems. We did this by using the Intensional View Displayer and choosing the number of entities in the extension as height of the view. There didn't appear to be any real problems except for the Actioned Structure Classes view and its dual view the Structured Action Classes which both became empty. When trying to understand the reason we found out that the view definition needed refinement. The introduction of some intermediate classes in version 1.90 forced us to use the `classInHierarchyOf` predicate instead of `subclassOf`.

Newly introduced views and relations

Because of the restructuring of the code in version 1.90, we needed to add one new view and one new relation:

Rendering Methods. The restructuring of the Execute Methods made us decide to define a new intensional view grouping all **Rendering Methods**.

Execute Methods *call* Rendering Methods The restructuring caused the responsibility of rendering web pages to be shifted from the execute methods to the rendering methods, but rendering was still triggered by the execute methods.

6 Experiment 3 (Verifying the design structure of *SmallWiki* 1.304)

In the third and last experiment we reverified our design documentation on yet a more recent version of SmallWiki (the one visually represented in Fig. 5) and drew conclusions about the usefulness of intensional views and relations to document the design structure of an evolving software system over a longer development period. Again, the design documentation appeared to be quite stable, but nevertheless we discovered some interesting inconsistent views and invalid relations, which are discussed next. We also compared the view sizes with those on the previous version.

Inconsistent Views

Outputtable Classes became inconsistent because of the addition of four new classes. Instead of having a specialized `accept*` method like the other `Outputtable Classes` (and as expected by the view definition), these classes delegated their accept method to a more general one. We solved this by refining the view such that it is declared as the conjunction of the classes with a specialized accept method together with the classes that delegate their accept method.

Storable Classes became inconsistent, as can be seen from Fig. 5, for the same reasons as the `Outputtable classes` view. By redefining this view in an analogous way, the consistency of this view was restored.

Execute Methods is no longer consistent because some coding conventions were not adhered to consistently in this version: there were two 'execute' methods that were not implemented in the correct protocol, and there were two other methods that were in the right protocol but did not start with the string 'execute'. To fix the problem the former just needed to be moved to the correct protocol whereas the latter either needed to be renamed or put in a more appropriate protocol.

Invalid Relations

Structure Classes *is subset of* Storable Classes and
Classes Visited *is subset of* Outputtable Classes failed because of the failure of the Storable Classes and Outputtable classes views, as discussed above. After fixing these views, these relations became valid again.

Storable Classes *are all accepted by* Store Visitors failed since the argument of the accept method on `LinkInternalVisitor` was called '`anInternalLink`' instead of on the expected '`aLinkInternal`'. (Remember that the predicate definition relied on the fact that the argument names respected a particular naming convention.) We fixed this problem by renaming the argument.

Store Visitors *all accept class of type* Storable Classes failed due to the addition of new storable classes which were not taken into account by the `Storable Classes` view. We solved this conflict by extending the `Storable Classes` view.

Output Visitors *all accept class of type* Outputtable Classes failed because in the original version of this intensional relation we documented the classes `AnObsoleteVisitorOutput` and `AnObsoleteVisitorHtml` as explicit deviations of the relation. These classes however were removed from the code between experiment 2 and 3 and thus caused this relation to fail. This was fixed by removing the deviating cases from the documentation again.

Comparing view sizes

We compared the sizes of the (extension of the) intensional views on version 1.90 with those on version 1.304 and observed two important differences: the number of Action Classes almost doubled (from 13 to 25), because more functionality had been added to *SmallWiki*, whereas the number of Execute Methods further diminished from 23 to 14, illustrating the continued migration from the old style of execute methods to those using the visitor pattern.

7 Critical analysis and lessons learned

In this section, based on our experiences gained with the *SmallWiki* case, we perform a critical analysis of the current generation of tools — including the new opportunities offered by the visualization tool — and of the underlying model of intensional views and

relations, to support co-evolution of high-level design and source code of a medium-sized *Smalltalk* application.

Deviations The experiments illustrated the importance of being able to define explicit deviations (inclusions and exclusions) to intensional views and relations. This happened when the implementation should have adhered to an intension or relation, but for various reasons did not. Typically, this either indicated an opportunity to refactor the code, or to refine an intension that was expressed too broadly. In either case it was useful to document the deviating cases explicitly. When eventually fixing the code or intension and reverifying consistency, the tool would issue warnings about deviations that had become obsolete, confirming us that the exceptional case had indeed been solved, at which point we could safely remove the corresponding deviation.

Completeness Although intensional views and relations allowed us to express and verify interesting structural constraints about the source code, the obtained design documentation was by no means complete. For example, it could prove useful to complement this design documentation with more dynamic information produced by other tools.

Static versus dynamic information Indeed, both query languages supported by our tool (*Soul* and *Smalltalk*) allowed us to define views and relations which reason about the *static structure of a system only*. Although we did not experience the lack of dynamic information as a severe hindrance while documenting the design of *SmallWiki*, we do agree that this restriction may prohibit us in documenting some interesting design constraints. For instance, the concept of a layered architecture is very hard to express without the use of dynamic information.

Incremental approach In our experiment, we adopted an incremental approach to document *SmallWiki*. Starting from a minimal working knowledge about the case, we gradually refined and documented our knowledge about the system by alternating manual code inspection with the definition of views and relations and verifying them against the source code. The tools helped us in codifying and testing our assumptions about the code structure and in finding out where the assumptions were (or became) invalid and why. This incremental approach not only allowed us to obtain a fine-grained documentation of the structure of *SmallWiki*, but at the same time helped us in obtaining a better comprehension of the system's implementation. In addition, we observed that verifying the documentation against newer versions of the code often provided us with valuable insights in how the application's design evolved.

Co-evolution The goal of the *IntensiVE* toolsuite is to support co-evolution of code and design documentation. To this end, our tools support the detection of structural conflicts between documentation and code, when either of them have evolved. We can discriminate between two kinds of conflicts. A first kind of conflict is when the documentation is conceptually correct, but some parts in the code violate it. This can happen when an actual bug was introduced in the code (e.g., removing a method that is being relied on) or when a certain naming or coding convention (e.g., putting a method in the wrong protocol) or architectural constraint was violated (e.g., adding a class that can be visited but does not implement the appropriate methods). In order to fix these conflicts, the code needs to be adapted. The other kind of conflicts that may occur are caused by code restructurings that affect the original design documentation. Such conflicts typically need

to be solved by modifying the design documentation, i.e. adapting the views and relations.

Perhaps surprisingly, the majority of conflicts we detected were of the second kind, i.e. they were caused by code restructurings of *SmallWiki*. Indeed, over the different versions of *SmallWiki*, the source code was often restructured in order to improve the design of the application. A possible explanation for the fact that we did not discover many conflicts of the first kind is that we did not apply the documentation to a system under development, but rather applied it 'a posteriori' to versions of *SmallWiki* which had already been released and tested.

Visualization One of the most recent additions to *IntensiVE* is the visualization tool. By making good use of the underlying *CodeCrawler* tool, we could use it not only to display the declared views and relations, but also to highlight inconsistent views and relations and to help us assess the impact of an evolution of the system. Using the *CodeCrawler* integration, with an appropriate metric we could for instance visualize the size of the views and the cardinality of the differences between the various alternatives of a view. This was a significant improvement over earlier versions of our tools where we had to manually inspect all views and relations in order to get an idea of the impact of evolution on the documentation. A downside of using the visualization was that, when the number of views and relations increased, the visual representation became cluttered. A pragmatical solution to this problem was to visualize only a selection of views and relations.

Choice of query language An interesting question when using *IntensiVE* is what query language to select. When defining an intensional view or relation, should we prefer logic queries over *Smalltalk* queries, or perhaps prefer hybrid queries? The rule of thumb we adopted was to always choose the language that best suited our needs, that is, the language in which we could express the query or predicate in the most compact, yet still declarative way. In practice, it often turned out that a hybrid query was most appropriate. For example, we could have defined the Structured Action Classes view by means of a logic query:

```
classWithName(?entity,?ename),
endsWith(?ename,['Action']),
classInViewNamed(?c,StructureClasses),
classWithName(?c,?cname),
equals([?cname, 'Action'], [?ename asString])
```

By using a mixture of logic and *Smalltalk* code, however, we could write the query much more compactly, by doing the string pattern matching in *Smalltalk* and the reasoning about the code structure in logic:

```
['*Action' match: ?entity name],
subclassOf(?c, [SmallWiki.Structure]),
[(?c name, 'Action') = ?entity name asString]
```

In an extreme case this even resulted in a hybrid query which took 4 lines of code, while the same query, written down in *Smalltalk* took 17 lines.

Nevertheless, without going in the technical details, when using *IntensiVE* we did occasionally notice some limitations when trying to mix queries and predicates defined in

the different languages. To solve these limitations, a better integration and symbiosis of the logic and *Smalltalk* query languages and libraries is required (like the one proposed in [11]).

Is logic programming needed? On the other hand, none of the declared views or relations in this case study required the full power offered by our logic programming language *Soul*. Hence we could probably use a less expressive but faster query mechanism like SmallLint [12], and still be able to codify the same views. But then we would also loose the abstraction facilities offered by our logic programming language, as well as its logic library containing an extensive set of predicates to reason about Smalltalk source code.

8 Conclusion

This paper investigated how the model of intensional views and relations and the *IntensiVE* toolsuite can be used to support co-evolution of source code and design of a software system. The evaluation was done by documenting the design of an early version of *SmallWiki* and checking this documentation against two more recent versions of *SmallWiki*. Doing these experiments we observed that:

- Although building a first version of the design documentation of an unknown system remains a largely manual process, the incremental nature of the approach, combined with tool support to verify and visualize conformance of the design against the code, helps us in understanding the code and its structure.
- Once the design of a system has been documented with intensional views and relations, conformance of this design against other versions can be checked and visualized. Even by simply reverifying the defined views and relations on another version of the software, we gain useful insights on how the software evolved.
- Visualization of high-level design documentation is useful and important, especially when combined with advanced metrics and coloring to highlight potential inconsistencies. In a glimpse of the eye it is possible to get an overview of the design, and assess whether it conforms to the code, and where not.
- Being able to use different query languages to express views and relations is important. It means that the language most appropriate to express certain kinds of information can be chosen. At the same time it reduces the learning cost of the approach: someone not proficient with logic programming can start with simple *Smalltalk* queries and gradually learn to use the logic language and library. A good integration and symbiosis of the query languages and libraries is essential, however.

Overall, despite some minor limitations of the environment, the *IntensiVE* toolsuite supported us quite well in documenting the high-level structure of *SmallWiki* and keeping it synchronized with the code as it evolved, while at the same time providing us with useful insights on how the code structure evolved over time.

References

[1] A. J. Ko, H. H. Aung, B. A. Myers, Eliciting design requirements for maintenance-oriented ides: A detailed study of corrective and perfective maintenance, in: Proceedings of the International Conference on Software Engineering ICSE'2005, IEEE Computer Society, 2005, pp. 126–135.

[2] K. Mens, R. Wuyts, T. D'Hondt, Declaratively codifying software architectures using virtual software classifications, in: Proceedings of TOOLS Europe 1999, IEEE Computer Society Press, 1999, pp. 33–45, TOOLS 29 — Technology of Object-Oriented Languages and Systems, Nancy, France, June 7-10.

[3] K. Mens, T. Mens, M. Wermelinger, Maintaining software through intentional source-code views, in: Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE'02), ACM Press, 2002, pp. 289–296.

[4] K. Mens, B. Poll, S. González, Using intentional source-code views to aid software maintenance, in: Proceedings of the International Conference on Software Maintenance (ICSM'03), IEEE Computer Society Press, 2003, pp. 169–178.

[5] K. Mens, A. Kellens, Towards a framework for testing structural source code regularities, submitted to ISCM 2005.

[6] R. Wuyts, S. Ducasse, Unanticipated integration of development tools using the classification model, Computer Languages, Systems and Structures 30 (1-2).

[7] K. Mens, I. Michiels, R. Wuyts, Supporting software development through declaratively codified programming patterns, Elsevier Journal on Expert Systems with Applications 23 (4) (2002) 405–431.

[8] M. Lanza, Codecrawler: Lessons learned in building a software visualization tool, in: Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003), IEEE Computer Society, 2003, pp. 409–418.

[9] L. Renggli, Collaborative web : Under the cover, Master's thesis, University of Berne (2005).

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Languages and Systems, Addisson-Wesley, 1994.

[11] K. Gybels, Soul and smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis, in: Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages, 2003.

[12] D. Roberts, J. Brant, R. Johnson, B. Opdyke, An Automated Refactoring Tool, in: Proceedings of ICAST 1996, Chicago, IL, 1996.

A New Object-Oriented Model of the Gregorian Calendar

Hernán Wilkinson
Mercap Development Manager
Tacuarí 202, 7mo Piso
C1071AAF, Buenos Aires, Argentina
54-11-4878-1118 (ext. 120)
h.wilkinson@mercapsoftware.com

Máximo Prieto
Lifia – Facultad de Informática
Universidad Nacional de La Plata
cc11, 1900, La Plata, Argentina
+54 221 422-8252 (ext. 215)
maximo.prieto@lifia.info.unlp.edu.ar

Luciano Romeo
Mercap Software Architect
Tacuarí 202, 7mo Piso
C1071AAF, Buenos Aires, Argentina
54-11-4878-1118
l.romeo@mercapsoftware.com

ABSTRACT
Time is an important aspect of all real world entities; therefore, temporal information is crucial in many computer-based applications. Different types of time entities exist such as those representing points in time and those representing measurements of time. Extensive research activity on temporal models has been done but the Smalltalk community has not benefited enough from them. Smalltalk-80 provides the classes **Date** and **Time** to model time domain entities. These abstractions cover the basic needs of most programs, but they are not enough when complex observations about time have to be programmed. ANSI Smalltalk added the **Duration** and **DateAndTime** classes**.** Squeak augmented the model with the abstractions **Timespan, Year, Month** and **Week**. While the Squeak model provides abstractions to cover almost all the observations within the time domain when using the Gregorian calendar, it lacks some abstractions and it does not properly model the problem domain. In this paper, we present a new set of classes that model entities of the time domain using the Gregorian calendar based on a simple metaphor. This model proved to be very powerful and easy to use. It allows programmers to design and program time related issues better than current time domain implementations, and in a more natural way.

Key words: Smalltalk, Date, Time, Gregorian calendar, Time span, Time intervals, Time line view, Relative Dates, Test Driven Development

1. INTRODUCTION
Time entities are an important aspect of many computer applications. For example, the financial domain has a strong coupling with the time domain because the value of any financial instrument is related to a certain point in time (i.e. the value of one Euro today is not the same as it was two years ago), financial operations among traders could be settled some time after a given date (i.e. 48 hours after today), instrument cash flows depend on dates relative to a certain calendar, and so on. Office information systems depend on time information to pay salaries, allow employees to leave on vacation, etc. Real time systems base their behaviour on timed events, verify the temporal evolution of the environment they control, etc.

Different types of temporal entities exist, such as:

- Specific points in the timeline, such as 01/01/2005 (defined as anchored data by [13])
- Measurements of time, such as 1 day (defined as unanchored temporal information by [13])
- Temporal information about occurred events, such us "John played his guitar while Paul was outside" ([2])

Many time models have been proposed in the past ([4], [13], [24], [19], [11]) but none of those models are provided within the Smalltalk environments. Also, most of them are related to other technologies such as relational databases or artificial intelligence systems. Works such as [8] and [15] propose changes to the ODMG [10] object model adding temporal tracking to objects, but they do not augment the ODMG time model which lacks important time abstractions. Other programming languages such as Java [16] and .NET [20] provide basic time models that suffer from important design flaws.

Barbic et al. in [26] and [27] classifies temporal systems in two categories, those that model *Time Representation* and those that model *Time Reasoning*. The former deals with the "*representation*" of time entities (time points vs. time intervals), *time ordering* (linear, circular or branching), *time boundedness* (i.e. modelling of finite or infinite times) and *time measurements* (distance between time entities, arithmetics on those measurements). The later focus on the specifications of a *time calculus* to manage temporal information and a *query language* to extract temporal information about time events.

We present in this paper an object model that focus on the *Gregorian calendar Time Representation*, implemented with Smalltalk, which provides abstractions for many of the time domain entities that are not model in current implementations.

1.1 Motivation
Our daily work focuses on financial applications, where temporal information is highly tied to the financial one. When we started to build financial applications with Smalltalk we realised that time objects provided by the environment were not enough to undertake the modelling of the financial domain.

Smalltalk-80 [12] provides a basic time model implementation of the Gregorian calendar. That model has not covered our expectations mainly because:

- It lacks proper abstractions of some important time domain entities (i.e. month, day)
- Time objects are not immutable (i.e. Time) therefore, they do not properly model time entities as we show further on.

The Chronology package [21] released with Squeak 3.7 [23] addresses many of the issues we found with the Smalltalk-80 model, but:

- It lacks a good separation between anchored and unanchored time entities
- It does not model important time entities such as month (i.e. January) and day (i.e. Monday).

The model we present in this paper is based on a simple metaphor and some modelling rules we outline further on. The metaphor proposes to see time entities as points of the time line with different resolution. Based on this metaphor, the model provides behaviour to:

- Determine which point comes before or after another (ordering of time points along a time-line).
- Go from one point in the time line to another.
- Obtain the distance between two time points.
- Switch from one scale to another.
- Represent segments of the time line of any scale.
- Represent intervals between points.
- Obtain views of the time line with certain filtering rules

The model also implements abstractions such as day, month, day of a given month and relative day among others. Another important characteristic of this model is that it uses Measurements [25] to represent the distance between two points in the time line, not just numbers as is commonly done in other models.

1.2 Scope

The model was developed out of a "commercial" necessity. Before creating this model, we looked for similar solutions in the Smalltalk community but none of them satisfied our needs. We decided to create a new model based on the exploration of the time domain using Test Driven Development [6] as the guidance technique.

The scope of the model is limited to the Gregorian calendar decreed by Pope Gregory XIII [22]. No support is given neither for the Hindu calendar nor for the Iranian one or any other calendar, see [22] for a complete description of these calendars. The model does not cover time entities that represent relations between events (i.e., "while", "before", "at the end", etc.).

1.3 Paper organization

The remaining of this paper is organized as follows: Section 2 expands the problem we present in this paper. Section 3 presents the metaphor we based the model on. Section 4 discusses the model's design and behaviour. Section 5 sketches the implementation. Section 6 compares the presented model with other time related models. Finally, Section 7 concludes the paper and gives directions for future research.

2. THE PROBLEM

Smalltalk-80 provides two classes to model time entities: **Date** and **Time**. These classes are subclasses of **Magnitude,** so their instances can be compared using the message #< (among others).

Class **Date** provides protocol to get the number of days between two dates (#*subtractDate*:) and to obtain a new **Date** by adding or subtracting a number of days (#*addDays:* and #*subtractDays:*). It also provides accessing protocol to get the year, month and day of an instance of **Date**. Although this abstraction is useful for many applications we encountered problems when dealing with complex situations like getting the number of months between two dates.

Some issues can be observed with the **Time** class as well. Instances of **Time** can only be created using a number of seconds from hour zero. No standard protocol is provided to create a **Time** instance with a number of hours, minutes, and seconds. If the programmer wants to do that, an instance of **Time** has to be created and the message #*hours:minutes:seconds:* has to be sent to the newly created instance. This message permits the modification of an object representing a time of the day, while our observations of reality made us conclude that time entities are immutable as we shall see in the following sections.

The Smalltalk-80 model also lacks abstractions to represent other entities found in the time domain such as years, months, days of a given month and some of them are confusing (i.e. **Time** behaves like a clock, not as a measurement of time). For instance, the message #*year* implemented in **Date** returns a **Number** not an object that reifies an entity "year". The same is also true with the message #*day*, it returns a **Number** representing the day number not a "day". To obtain the month of a **Date** it is even harder because the model does not have a month class. **Date** provides two messages to accomplish that requirement, #*monthName* and #*monthIndex*. The former returns a **Symbol** (i.e. #February) and the later a **Number** representing the position of that month in a Gregorian year (i.e. 2 for February).

It could be argued that these are subtle issues, that a day can be modelled as a **Number** and a month can be modelled as a **Symbol** or as a **Number**. An example of such model is the one provided by Smalltalk-80. We argue that a better model can be created because this implementation lacks abstractions which make it difficult to use when complex time-related calculations and situations need to be programmed.

For instance, the Smalltalk-80 model does not easily solve the problem of getting the number of days of a month because the object that represents a month is a **Symbol** or a **Number** and neither of them answers the message *#numberOfDays*. Class protocol is provided in **Date** to answer that question with the message *#daysInMonth: aMonthName forYear: anInteger* but we argue that the class **Date** should not be responsible for this behaviour. A better solution would reify the "month of year" concept providing to this abstraction the necessary behaviour to treat it as a month of year, not a **Symbol** or a **Number**, with messages such as *#numberOfDays*. (See Figure 1)

```
"<<<< Smatalk-80 Solution >>>>
Note that the message #daysInMonth:forYear: is sent to the class Date"
today := Date today.
Date daysInMonth: today monthName forYear: today year.
```

Figure 1: Getting the number of days of a year's month

Squeak version 3.7 provides a richer model with abstractions proposed by the ANSI Standard [3] like the class **DateAndTime** and the class **Duration**, used "to represent a length of time" [3]. It also reifies concepts like **Timespan**, **Year**, **Month** and **Week**, implemented as subclasses of **Timespan**.

The Squeak model, although richer than the Smalltalk-80 and the ANSI models, also lacks abstractions to represent a day, a day in a month or just a simple month. It can at first produce misinterpretations on the meaning of its abstractions such as the class **Month,** which does not represent a month (i.e. January) but a month in a year (i.e. January 2005). But the main problem we found with this model is that time entities are modelled as segments in the time line; all the time classes are subclasses of **Timespan**. This modelling decision merges two different concepts, time points and time segments in one, which allows comparing entities of different granularity such as years and dates (i.e. year 2005 and January 2nd of 2005).

The problem with representing time entities as time segments is that a total order can not be defined among them (See [13]). Therefore, the result of comparing those entities could be "unknown" (i.e. year 2005 is not less, equal or greater than January 2nd of 2005) and the "unknown" entity is not modelled in Squeak.

Due to the limitations of the existing models shown in this section we decided to create a new model of the Gregorian calendar reifying as much time entities as we observed from reality.

3. THE METAPHOR

We use a metaphor to understand the time domain. In this metaphor, time entities are points in a line, a line that represents the time line. The observers of that line can zoom in and out the points it contains. When the observer zooms in she sees smaller points (i.e. dates), when the observer zooms out she sees bigger points (i.e. years). We say that the time line has different scales or that time lines of different scale can represent the passing time.

Let's see an example. A year represents a point in time but with less resolution than a date. If the year is zoomed in, new points will be observed; those points are the months of that year. If one of those points is picked and zoomed in, the points representing the dates of that month will be obtained. If one of this dates is selected and zoomed in, points representing the hour of that date will be obtained. Let's do it with concrete entities. If the year 2005 is selected and zoomed in, months from January of 2005 to December of 2005 will appear. If January of 2005 is zoomed in, dates from January 1st of 2005 to January 31st of 2005 will be seen. If January 1st of 2005 is zoomed in, the entities January 1st of 2005 at 00:00:00 to January 1st of 2005 at 23:59:59 will be seen. See Figure 2 for a graphical representation.

The inverse happens when zooming out. If an hour of a day is zoomed out, a point representing its date will be obtained. If that date is zoomed out, a point representing the month where that date belongs to will be obtained. If that point is zoomed out, the year that the month belongs to will be obtained.

The points that can be obtained at the different scales of the time line are abstractions representing years (i.e. year 2005), months of a year (i.e. January of 2005), dates (i.e. 01/01/2004) and the time of a given date (i.e. 01/01/2005 at 00:00:00).

Even though these are the only kinds of points we can obtain from the time line (at least in our model), there are other entities that we also modelled, such as the days of the week (i.e. Monday), days in a month (i.e. January 1st), hours in a day (i.e. 00:00:00), months (i.e. January), segments of the time line and relative dates among others.

4. PROPOSED MODEL

As we said before, the main drawback of the Smalltalk-80 and Squeak models is that they do not provide abstractions for all the entities that we can observe in the time domain related to the Gregorian calendar. Because software is knowledge represented in a computable model, object models should provide an abstraction for each observed entity of the problem domain. Lacking abstractions means incomplete knowledge. Incomplete knowledge leads programmers to fill the gaps between the problem domain and its model with solutions that end up producing code duplication, ad-hoc implementations and finally, error prone situations. Object models with the right abstractions are more reusable and easier to use.

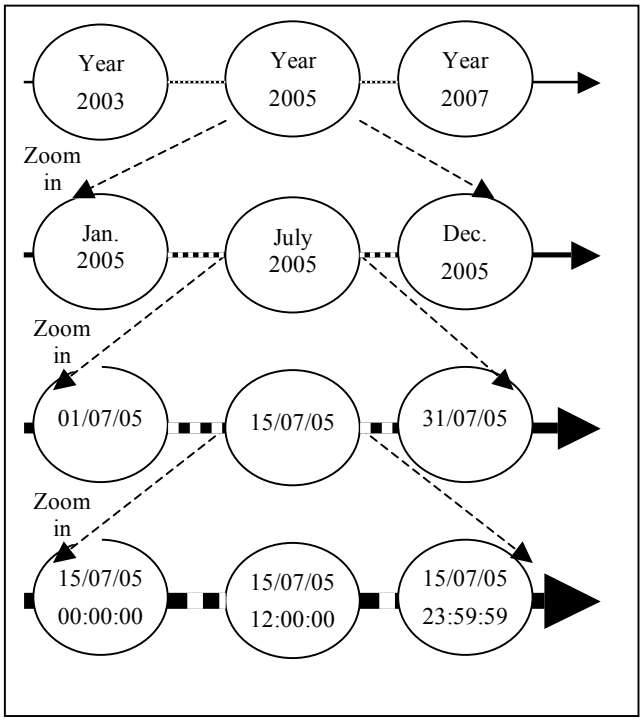

Figure 2: Zooming in and out in the time line

Based on this principle, we observed and modelled the following entities of the Gregorian calendar:

- **Years**: Modelled with the class **GregorianYear**. This class is used to represent years such as the year 2005, the year 2000, etc.
- **Months of a Year**: Modelled with the class **GregorianMonthOfYear**. This class represents entities like January of the year 2005, December of the year 2000, etc.
- **Dates**: Modelled with the class **GregorianDate**. It is used to represent entities such as 31/12/2005, which is December 31st of 2005. Note that we use the DD/MM/YYYY notation.
- **Relative Dates**: Modelled with the class **RelativeGregorianDate**. Used to represent dates that can change depending on different time events (i.e. working or none working days).
- **Time of a given Date**: Modelled with the class **GregorianDateTime**. This class represents entities such as 01/01/2005 at 10:00:00, that is, ten in the morning of January 1st of the year 2005.
- **Days of a Month**: Modelled with the class **GregorianDayOfMonth**. This class represents entities such as January 1st, December 25th, etc. Note that these are days of given months but of no particular year.
- **Months**: Modelled with the class **GregorianMonth**. Months are January, February, March, etc.
- **Days**: Modelled with the class **GregorianDay**. Days are Sunday, Monday, Tuesday, etc.
- **Time of a Day**: Modelled with the class **TimeOfDay**. It represents the time in a day such as 10 AM, 12 PM, 9:15:35 (this is quarter past nine and thirty five seconds).
- **Segments of the time line**: Modelled with the class **Timespan** (i.e. 10 days from now)
- **Time point intervals of different granularity and resolution**: Modelled with the class **MeasurementInterval** (i.e. from 01/01/2005 to 20/01/2005 every 3 days).
- **Time line views**: Modelled with the class **TimelineView**. Used to mark time points according to some criteria (i.e. working day, non working day).

4.1 Time entities immutability and validity

Something we have noticed about time entities is that they are immutable; they do not change, they are immutable like the numbers. A given date such as *January 1st of 2005* should not allow its year, month or day to be changed. Therefore, the abstractions we use to model the time domain entities are immutable, they behave like "value objects" (see [5]). Immutable objects allow us to have a simpler model and not to worry about inconsistent objects, invalid modifications or invariance invalidity during a certain time.

The model also verifies, when creating an object, if the new instance will be valid. If that is true, the object is created, otherwise an exception is signalled. Therefore, the code that verifies if an object is valid is located in one place and ensures that no invalid time objects exist.

For example, the year zero is an invalid year on the Gregorian calendar, and trying to create an object for the year zero is a semantic error, so we check that rule when trying to create an instance of **GregorianYear**. See Figure 3.

```
GregorianYear class>>number: aNumber
^(self isValidYearNumber: aNumber)
  ifTrue: [ ... create the instance ...]
  ifFalse:[InvalidGregorianYearNumberException signalNumber: aNumber ].

GregorianYear class>>isValidYearNumber: aNumber
  ^aNumber~=0 and: [ aNumber isInteger ]
```

Figure 3: Verifying the creation of an instance of a year

Because **GregorianYear** is immutable, no instance message is provided to set the number of the year. If **GregorianYear** were not immutable, the setter method *#number:* would have to perform the same verification as the *#number:* class method. This verification is not difficult to do with years, but what about dates? If we provide a message to change the day number, its implementation should verify that the day number is valid for the month and year the date already represents. But, what happens if it is temporarily invalid because the next collaboration modifies the month making the new day number valid? There is no way to maintain the validity of the date invariants if we provide messages to modify its day number, month or year.

A message could be provided to completely change a date such as *#yearNumber: aYearNumber monthNumber: aMonthNumber dayNumber: aDayNumber*, but that message would be the same as that one sent to the class to create a new instance as Figure 4 shows.

```
"Creates the date 28/2/2005"
aDate := GregorianDate yearNumber: 2005 monthNumber: 2 dayNumber: 28.
"Setting the day number to 31 should signal an exception"
aDate dayNumber: 31.
"But if the month is changed to be January the previous day number would be valid..."
aDate monthNumber: 1.
"A message to change the year, month and day number could be provided, but it is the same as the one the class responds to"
aDate yearNumber: 2005 monthNumber: 1 dayNumber: 31
```

Figure 4: Verifying the creation of an instance of a year

4.2 Different scale time line traversal

As we said before, a year can be seen as a point in the time line at a year resolution. Because the resolution is a year, that point contains other points of higher resolution such as months of a year, dates and time in a certain date. The model provides protocol to easily move between points of different resolutions (i.e. going from a year to the dates it contains or from a date to its year). Moving to points of smaller resolution looks natural (i.e. going from a date to its year) but moving to points of higher resolution is not so commonly provided on this type of models (i.e. going from a year to its dates).

Messages to go from points of one scale to another are provided on each abstraction. See Figure 5 for an example.

```
aYear := GregorianYear number: 2005.
"Going from years to months of year"
aYear firstMonth. "Returns January of 2005"
aYear lastMonth.  "Returns December of 2005"
aYear months.     "Returns all the months of year 2005"
"Going from years to dates"
aYear firstDate   "Returns 01/01/2005"
aYear lastDate    "Returns 31/12/2005"
aYear dates       "Returns the 365 dates of the year 2005"
aYear firstDay    "Returns Saturday"
aYear lastDay     "It is also a Saturday"
"Going from years to date times"
aYear firstDate atMidnight   "Returns 01/01/2005 00:00:00"
aYear lastDate lastTimeOfDay "Returns 31/12/2005 23:59:59"
```

Figure 5: Moving from a year to other entities

4.3 Magnitude protocol

All the time point abstractions respond to the magnitude protocol with messages such as *#<, #<=, #>, #>=, #min:, #max:, #between: and:* among others. Because they are points in the time line of a certain resolution, they can be compared to see which one is closer or farther from the beginning of the time line. A total order can be defined for them. See Figure 6.

```
(GregorianYear number: 2005) < (GregorianYear number: 2010)                    "Comparing years"
GregorianMonthOfYear decemberOf: 2005 < GregorianMonthOfYear julyOf: 2005      "Comparing month of year"
GregorianDate today < GregorianDate tomorrow                                   "Comparing dates"
GregorianDateTime now < GregorianDateTime now next                             "Comparing datetimes"
```

Figure 6: Comparing points on the time line

Not only points on the time line can be compared. Instances of **GregorianDay**, **GregorianDayOfMonth** and **GregorianMonth** can also be compared. When comparing days of the week, the model assumes Sunday is the first day of the week but this can be changed to any other day such as Monday. January 1st is always the first **GregorianDayOfMonth** and January is always the first **GregorianMonth**. Figure 7 shows how to compare these objects.

```
GregorianDay monday < GregorianDay tuesday                                     "Comparing days"
GregorianMonth january < GregorianMonth december                               "Comparing months"
'01/01' asGregorianDayOfMonth < '25/12' asGregorianDayOfMonth                  "Comparing days of month"
```

Figure 7: Comparing other time entities

Comparing points of different resolution can end up being "unknown". For example, the year 2005 is not less, equal or greater than January 2nd of 2005. Different approaches were proposed to solve this problem. [13] and [4] propose to return "unknown" for this type of comparison. Squeak does not return unknown but it can be inferred because all the comparison messages (#<,#= and #<) return *false* when they are sent to objects under this situation. We propose a different solution where the comparison between points of different resolutions is not allowed and, if such an attempt is made an exception is signalled.

This decision is based on the metaphor used to create the model and an analogy we made with points and sets. Because points in the time line are composed of other points, they can be considered analogous to sets. For example, a year is a point that contains the months of that year. We think that comparing a year (seen as a set of its months) with a month of that year (an element of that set) is a semantic mistake because it is analogous to compare a set with elements of that set.

Propositions such as "Is the year 2004 before January 1st of 2005?" are seen as valid because only a comparison at the year resolution is necessary to answer that question, only the year 2004 and the year 2005 are compared. The problem with this type of comparison arises when comparing a year with a month of that same year such as "Is the year 2005 before March of 2005?" Because March of 2005 is part of the year 2005, it is neither before, after nor equal to the year 2005, but included in it.

4.4 Obtaining the distance between two points

Time models should provide ways to know the number of years between two years, the number of months between two months of a year, and so on. This is analogous to obtain the number of points between two points of the same time line resolution.

Messages *#distanceTo: aPoint* and *#distanceFrom: aPoint* are used to obtain the distance between two points. The same messages are used polymorphically for years, months of a year, dates, etc. The model does not provide the message #- (minus) to get the distance between two points because it does not behave like the subtraction operation. When the message #- is sent to a **Number**, it returns another **Number**, but the distance between two points in the time line is not of the same type of the points; it is a measurement. Due to this observation we decided to use a different protocol for this kind of inquires. See Figure 8.

The model also provides behaviour to obtain the distance between time entities like days, months and days of months.

```
(GregorianYear number: 2005) distanceTo: (GregorianYear: 2010)          "Returns 5 years"
(GregorianYear number: 2005) distanceTo: (GregorianYear: 2000)          "Returns -5 years"

'01/01/2005' asGregorianDate distanceTo: '10/01/2005' asGregorianDate   "Returns 10 days"
'01/01/2005' asGregorianDate distanceFrom: '10/01/2005' asGregorianDate "Returns -10 days"
```

Figure 8: Getting the distance between two points

4.5 Time measurements and their relevance on the time domain

Note that objects returned by the distance messages are not numbers but time measurements. Some models provide abstractions for such entities like [4] and [13], others just do not reify them like Smalltalk-80 and Squeak, where raw numbers are used to represent them. This model reifies them reusing another model we created, one used to represent any kind of measurement. In such model, a measurement is modelled as a number together with a unit.

The advantages of using measurements over raw numbers are explained in [25], [1] and [17]. We would like to briefly mention some of them. The first and most important one is that the object "*10 days*" represents in a better way the distance between days than just the number "*10*". People could argue that in reality, when they are asked how many days there are between two dates, i.e. how many days are between January 1st and January 10th, they just respond with a number, i.e. 9. That is true, we "say" a number but that number has implicit knowledge attached to it due to the context of the question that has been asked. Its meaning is not just *9*, but *9 days*.

This model provides different units to create all the possible measurements of the Gregorian calendar. These units are organised in two different categories due to the irregularity of the Gregorian calendar. The base unit for each category is *month* and *millisecond*. Figure 9 shows the units provided by default with the model, new units can be created.

Unit	Type	Measurement example	Conversion example
month	Base Unit	10 months	(12 months convertTo: year)=1 year
year	Derived from month	2 years	(2 years convertTo: month)=24 months
decade	Derived from month	1 decade	(1 decade convertTo: year)=10 years
century	Derived from month	2 centuries	(2 centuries convertTo: decade)=20 decades
millennium	Derived from month	1 millennium	(1 millennium convertTo: century)=10 centuries
millisecond	Base Unit	1000 milliseconds	(1000 milliseconds convertTo: second)=1 second
second	Derived from millisecond	60 seconds	(60 seconds convertTo: minute)=1 minute
minute	Derived from millisecond	60 minutes	(60 minutes convertTo: hour)=1 hour
hour	Derived from millisecond	24 hours	(24 hours convertTo: day)=1 day
day	Derived from millisecond	7 days	(7 days convertTo: week)=1 week
week	Derived from millisecond	2 weeks	(2 weeks convertTo: day)=14 days

Figure 9: Time units provided by default

Note that converting measurements of different scales is not always feasible due to the irregularity of the Gregorian calendar. [13] also explains this limitation. In this model, measurements can be automatically converted if they share the same base unit. A measurement of years can be converted to months, decades, centuries and millenniums because they share the same base unit, *month*. Automatic conversion between milliseconds, seconds, minutes, hours, days and weeks is also possible because they share the same base unit, *millisecond*.

A measurement of years cannot be converted to days because the conversion could be *366 days* or *365 days* per year due to the existence of leap years in the Gregorian calendar. The same applies to months. A month cannot be converted to days because it could represent *28, 29, 30* or *31 days*. This does not mean that a specific year or month of year can not be asked for the number of days it contains. Instances of **GregorianYear** and **GregorianMonthOfYear** respond to the message #*numberOfDays*, which returns a time measurement (i.e. *29 days* if the month of year is February 2004 and *28 days* if the month of year is February of 2005).

Because the time model uses the measurement model, new time units can be created as needed. For example, the *quarter* of a year unit can be created as derived from *month* as shown in Figure 10.

```
month := BaseUnit nameForOne: 'month' nameForMany: 'months'  "This unit is provide with the model"
quarter := DerivedUnit from: month
                    nameForOne: 'quarter' nameForMany: 'quarters'
                    convertionFactor: 3
```

Figure 10: Creating a new time unit

It is also possible to mathematically operate with time units because the measurement model provided with this model supports the basic arithmetic operations +, -, * and / among others. Because time units are reified, measurements composed with time measurements can be created, such as *100 Km/hour* (a measurement of speed) or *10%/month* (an interest rate of 10% by month). Figure 11 shows some examples. Refer to [25] for a complete explanation of this behaviour.

```
14 days + 1 week = 1814400000 milliseconds.                    "Adding measurements of the same base unit"
((14 days + 1 week) convertTo: days) = 21 days.                "Converting the result of an operation"
(1 year + 10 days) = (1 year + 10 days)                        "Adding measurements of different base unit"
10 years * 10 = 100 years                                      "Multiplying a measurement by a number"
10 years * 12 months = 10 year*year                            "Multiplying measurements"
10 years * 12 months / 24 months = 5 years                     "The model automatically simplifies units"
100 kilometers / 1 hour                                        "Represents a speed of 100 km per hour"
0.01 / 1 month                                                 "Represent an interest rate of 10 % by month"
```

Figure 11: Arithmetic with time measurements

4.6 Moving through points of the same time line resolution

The model provides the #*next*, #*next: aMeasurement*, #*previous* and #*previous: aMeasurement* messages to move certain distance from a given point. #*next* and #*previous* messages assume that the distance to move is equal to the quantum of the time line the point receiving the

message belongs to. If the point is a year, the quantum is *1 year*, if the point is a month of a year the quantum is *1 month*, if the point is a date the quantum is *1 day* and if the point is a date time the quantum is *1 millisecond*.

Moving certain distance from a point expects a measurement of time as parameter because the distance between two points is expressed as a measurement of time. See Figure 12 for examples.

(GregorianYear number: 2005) next ***"Returns GregorianYear number: 2006"***
(GregorianYear number: 2005) next: 1 year ***"Returns GregorianYear number: 2006"***
(GregorianYear number: 2005) next: 12 months ***"Returns GregorianYear number: 2006"***
(GregorianYear number: 2005) next: 10 years ***"Returns GregorianYear number: 2015"***
(GregorianYear number: 2005) previous: 5 years ***"Returns GregorianYear number: 2000"***

Figure 12: Moving on the same time line resolution

At the moment this paper was written moving a certain distance expressed in a unit not convertible to the unit of the quantum of the point signals an exception. We found this behaviour to be too restricted when dealing with some financial observations. In the section future work we show some ideas to solve this problem. Figure 13 shows examples of how the model behaves at the time this paper was written.

(GregorianYear number: 2005) next: 120 days ***"Signals an exception because 120 days can not be converted to years"***
'01/2005' asGregorianMonthOfYear next: 120 days ***"Signals an exception because 120 days can not be converted to months"***

Figure 13: Moving on the same time line resolution

The model also provides protocol to move through time entities that do not belong to any time line but have an order such as days, months and days of month. See Figure 14.

GregorianDay monday next: 4 days ***"Returns Friday"***
GregorianMonth january next: 2 months ***"Returns March"***
(GregorianMonth january dayNumber: 1) next: 2 days ***"Returns January 3rd "***

Figure 14: Moving from days, months and day of months

4.7 Segments of the time line

The class **Timespan** represents segments of the time line. A segment begins on a specific point of the time line and has certain duration and direction expressed as a measurement. The starting point of a time span can be a point at any of the time line resolutions. The duration and direction is given by a time measurement that should be convertible to the unit of the scale the staring point belongs to. If the measurement is positive, the direction is towards the end of time, if the measurement is negative, the direction is towards the beginning of time. See Figure 15.

"Creates a time span from January 1st of 2005 with 72 hours of duration"
aTimespan := Timespan from: '01/01/2005' asGregorianDate duration: 72 hours.
aTimespan to. **"Returns 4/01/2005"**
"Creates a time span from year 2005 with a duration of 4 years"
aTimespan := Timespan from: (GregorianYear number: 2005) duration: 4 years
aTimespan to. ***"Returns year 2009"***
"Creates a time span from now with a length of 3 weeks toward the beginning of time"
aTimespan := Timespan from: GregorianDateTime now duration: -3 weeks
aTimespan to. ***"If now is 01/01/2005 10:00:00, returns December 11th of year 2004 at 10 AM"***

Figure 15: Time spans of point in lines

Time spans can also be used with time objects that are not part of the time line but have an order such as days, months and day of months. Figure 16 shows some examples.

(Timespan from: GregorianDay today duration: 3 days) to. ***"Returns Thursday if today is Monday"***
(Timespan from: GregorianMonth current duration: 6 months) to. ***"Returns July if the current month is January"***

Figure 16: Time spans of days, months and day of months

Time spans are useful to represent relative time entities where the beginning of such an entity is known, but the end is not exactly known or can change. Examples of such entities are "I'll see you in 10 working days from today" or "it happened 7 months before January". Time spans are important to represent relative time entities such as relative dates which are explain further on.

4.8 Intervals

The model reifies the concept of intervals for time entities with an order. Those intervals behave like collections between the specified starting and ending point. Measurements are used to specify the step of those intervals.

The same protocol used to create intervals of numbers is used to create intervals of time entities. For example, an interval between two years can be created sending the message *#to:anotherYear by: aDistance* to an instance of **GregorianYear**. See Figure 17.

"Returns an Interval with eleven elements, the years between 2005 and 2015 inclusive".

(GregorianYear number: 2005) to: (GregorianYear number: 2015)

"Returns an Interval with six elements, the years 2005,2007,2009,2011,2013 and 2015 inclusive".

(GregorianYear number: 2005) to: (GregorianYear number: 2015) by: 2 years

"Returns an Interval with six elements, the years 2005,2004,2003,2002,2001 and 2000 inclusive". (GregorianYear number: 2005) to: (GregorianYear number: 2000) by: -1 year

Figure 17: Interval creation

Time intervals are polymorphic with number intervals, which at the same time behave as collections. Figure 18 shows some examples.

"Returns all the leap years between 2005 and 2100"

((GregorianYear number: 2005) to: (GregorianYear number: 2100)) select: [:aYear | aYear isLeap]

"Returns all Sundays between January 1st of 2005 and the last date of February 2005"

('01/01/2005' asGregorianDate to: '02/2005' asGregorianMonthOfYear lastDate) select:

[:aDate | aDate isSunday]

Figure 18: Using intervals

The model also provides protocol to create collection of objects that are commonly used. See examples of Figure 19.

"Returns all the Tuesdays between January 1st of 2005 and June 30th of 2005"

'01/01/2005' asGregorianDate to: '30/06/2005' asGregorianDate everyDay: GregorianDay tuesday

"Returns all dates whose day number is 10 between January 1st of 2005 and June 30th of 2005"

'01/01/2005' asGregorianDate to: '30/06/2005' asGregorianDate everyDayNumber: 10

"Returns all dates whose day numbers are 10 or 20 between January 1st of 2005 and June 30th of 2005"

'01/01/2005' asGregorianDate to: '30/06/2005' asGregorianDate everyDayNumbers: #(10 20)

Figure 19: Commonly used protocol

The difference between time intervals and time segments is subtle. Time intervals are collections while time segments are not. Time segments can not be iterated and they are not composed by a collection of time entities, they just have a beginning and a directed duration. Protocol to convert from a time interval to a time segment and vice versa is provided by the model.

4.9 Time line views

The model reifies the concept of time line view. A view behaves as a filter of a certain time line universe restricting the elements that belong to that universe. Views are defined by a collection of rules.

A common use of such view is to filter working and non working days. For example, a view can be created to mark all Saturdays and Sundays as non working days, another view can be created to filter the months where the season changes, etc..

The model provides different types of rules, such as a rule for days (i.e. to include all Saturdays), a rule for a given day in a month (i.e. all the 25th of May), a rule for specific time entities and different rule decorators.

Views behave like collections, so they can be iterated, they can be query for the inclusion of elements, etc. Figure 20 shows how to create a view for non working days.

```
"Let's create a view for all dates..."
nonWorkingDaysView := TimelineView universe:
  (GregorianDate theBeginningOfTime to: GregorianDate theEndOfTime).
"Now, we want Saturdays to be on that view"
nonWorkingDaysView addDayRule: GregorianDay saturday.
"Now we want Sundays from  January 1st of year 1000 to the end of time..."
nonWorkingDaysView addDayRule: GregorianDay sunday
  from: '01/01/1000' asGregorianDate to: GregorianDate theEndOfTime.
"Now we want all July 9th since 1816 because  is the Independence Day in Argentina".
nonWorkingDaysView addDayOfMonthRule: '9/7' asGregorianDayOfMonth
  from: '9/7/1816' asGregorianDate to: GregorianDate theEndOfTime.

nonWorkingDaysView includes: '9/7/2005' asGregorianDate      "Returns true"
nonWorkingDaysView includes: '8/7/2005' asGregorianDate      "Returns false"
nonWorkingDaysView includes: '16/7/2005' asGregorianDate     "Returns true, it is Saturday"
nonWorkingDaysView includes: '17/7/2005' asGregorianDate     "Returns true, it is Sunday"
nonWorkingDaysView includes: '18/7/2005' asGregorianDate     "Returns false, it is Monday"
```

Figure 20: Time line views

Views can be really vast and impossible or too slow to iterate on them. The model provides streams whose responsibility is to move through an interval of the elements of the view. Figure 21 shows an example

```
"Streams over the next 10 non working days, starting from today"
stream := TimelineStream from: GregorianDate today using: nonWorkingDaysView.
10 timesRepeat: [ stream next ]
```

Figure 21: Calendar streams

Because time line views are defined by rules, the inverse or negation of a view is easy to obtain. A negated view includes all the time entities that its original view excludes and vice versa. When the message *#negated* is sent to a view, its inverse is returned. As we shall see in the next section, negated views are important in the financial domain.

4.10 Relative Dates

In the financial domain, settlement dates are usually expressed as a distance from the trade date in a given calendar. For example, a trader can buy bonds on a Thursday, but the settlement date is set to happen within 48 hours using the clearing house's calendar. That usually means that the trader's institution will receive the bonds on the next Monday, but this is true only if that Monday is a working day and it could have been true at the time the operation was done. But sometimes non-working days are created due to non-expected events (i.e. the death of some important person) and a working day is declared to be non-working.

In our example, if Monday is declared as non-working day, the new settlement date for the trade will be Tuesday. To model this new type of entity we created an abstraction called **RelativeGregoriaDate** that is a date relative to a time line view given a certain time span. See Figure 22 for an example. Note that the settle date is declared using the negated non-working days view because settlements can occur only on working days.

```
"06/01/2005 is a Thursday"
aTimespan := Timespan from: '06/01/2005' asGregorianDate duration: 48 hours.
aSettleDate := RelativeGregorianDate timespan: aTimespan calendar:  nonWorkingDaysView negated.
nonWorkingDaysView includes: '10/01/2005' asGregorianDate "Return false because 10/01/2005, a Monday,
                                  is a working day"
aSettleDate absoluteDate.                    "Returns 10/01/2005"
"Now a new non working day is added to the view"
nonWorkingDaysView addDateRuleFor: '10/01/2005' asGregorianDate
nonWorkingDaysView includes: '10/01/2005' asGregorianDate "Return true. Now 10/01/2005, is a not
                                   working day"
aSettleDate absoluteDate.                    "Now it returns 11/01/2005 because the
                                  view has changed"
```

Figure 22: Relative dates

Relative dates change according to the changes on the view they are related to. Its instances are polymorphic with **GregorianDate**. Relative dates show the importance of reifying the time line segment. Because the absolute date represented by a relative date depends on a view, it has to be declared as a segment of a time line that is filtered with the view associated to the relative date.

4.11 Special time entities

The time line does not have a known end or beginning, but the mere fact that we, as human, can think on them means that they have to be reified. Two objects are provided to represent these entities. They are "*theEndOfTime*" and "*theBeginningOfTime*". The object "*theEndOfTime*" is always greater than any point in time and "*theBeginningOfTime*" is always less than any point in time.

These objects are useful to create open intervals towards infinite and minus infinite. They allow programmers to create intervals and views on the whole time line and to create streams with no end. When using these objects, the programmer has to have special care because iterating over an interval with no end and/or beginning will never stop.

5. MODEL'S IMPLEMENTATION

5.1 Points in Time

PointInTime is the class that represents the abstract concept of a point in the time line. It is the superclass of all the concrete points of the time line such as year, date, etc., and it provides common implementation to the shared messages. Two methods have to be implemented by its subclasses, #*next*: and #*distanceTo*:. Messages such as #*previous*: and #*distanceFrom*: are implemented using them. **PointInTime** is a subclass of **IntervalAwareMagnitude,** which is an abstract class that provides common protocol and implementation to create intervals.

In Smalltalk 80, messages such as *#to:, #to:by:* and *#to:by:do:* are only implemented by **Number.** We extended the responsibility of creating intervals to all magnitudes. These intervals are instances of **MeasurementInterval**, they can be used with any **Magnitude** and they are polymorphic with **Interval**. Before a new instance of **MeasurementInterval** is created, the validity of the future interval is verified, and if it is not valid an exception is signalled. See Figure 23.

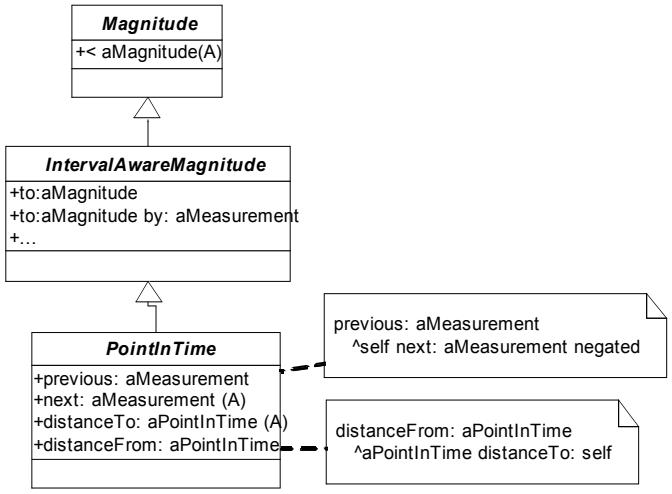

Figure 23: PointInTime abstract class

5.2 Years

The lack of uniformity of the Gregorian calendar has been modelled using classes to represent the special cases. For example, Gregorian years can be leap or non-leap, so there is a class representing leap years (**GregorianLeapYear**) and a class representing non-leap years (**GregorianNonLeapYear**). When the **GregorianYear** class receives the message #*number: aNumber* to create an instance of a Gregorian year, it verifies wether the number corresponds to a leap year or a non-leap year. If the number corresponds to a leap year it returns an instance of **GregorianLeapYear**, otherwise it returns an instance of **GregorianNonLeapYear**. The programmer should not care about a year's class, he just needs years to behave as expected.

Because leap and non-leap years are reified, no conditional statement has to be used to implement messages such as #*numberOfDays*. If the year is leap, the message #*numberOfDays* returns *366 days*, if the year is not leap, the message #*numberOfDays* returns *365 days*. See Figure 24.

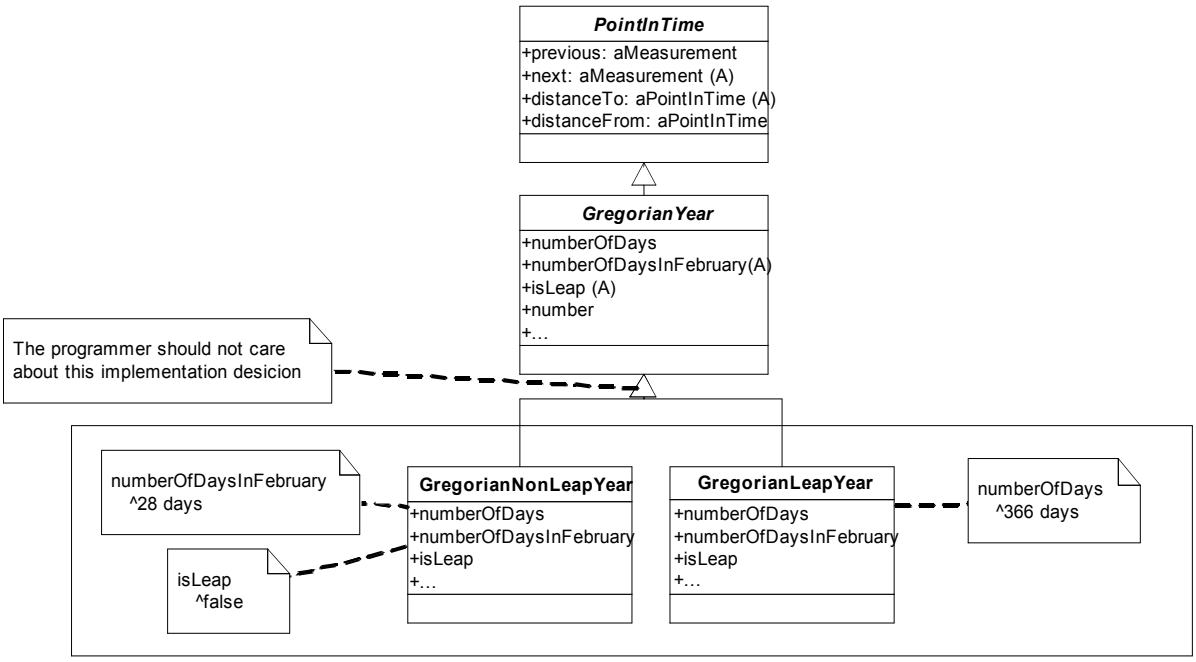

Figure 24: GregorianYear class hierarchy diagram

5.3 Months and Months of Year

February is another example of the lack of uniformity of the Gregorian calendar. Its number of days depends on the year. To solved this problem we modelled months with an abstract class named **GregorianMonth** and specific implementations such as **FebruaryGregorianMonth**, **JanuaryGregorianMonth** and **NonSpecificGregorianMonth**. Months are obtain sending messages to **GregorianMonth** such as *#january, #february, #march,* etc. Only one instance of each month exists. The programmer should not care about this implementation decision.

When a **FebruaryGregorianMonth** receives the message *#numberOfDaysIn: aGregorianYear*, it sends the message *#numberOfDaysInFebruary* to *aGregorianYear*. If that year is leap, it returns *29 days*, if it is non–leap, it returns *28 days*. Note that no conditional message has to be sent. When a **JanuaryGregorianMonth** receives the message *#numberOfDaysIn: aGregorianYear* it returns *31 days*. When a **NonSpecificGregorianMonth** receives that message it returns the object referenced by the instance variable *numberOfDays*. See Figure 25.

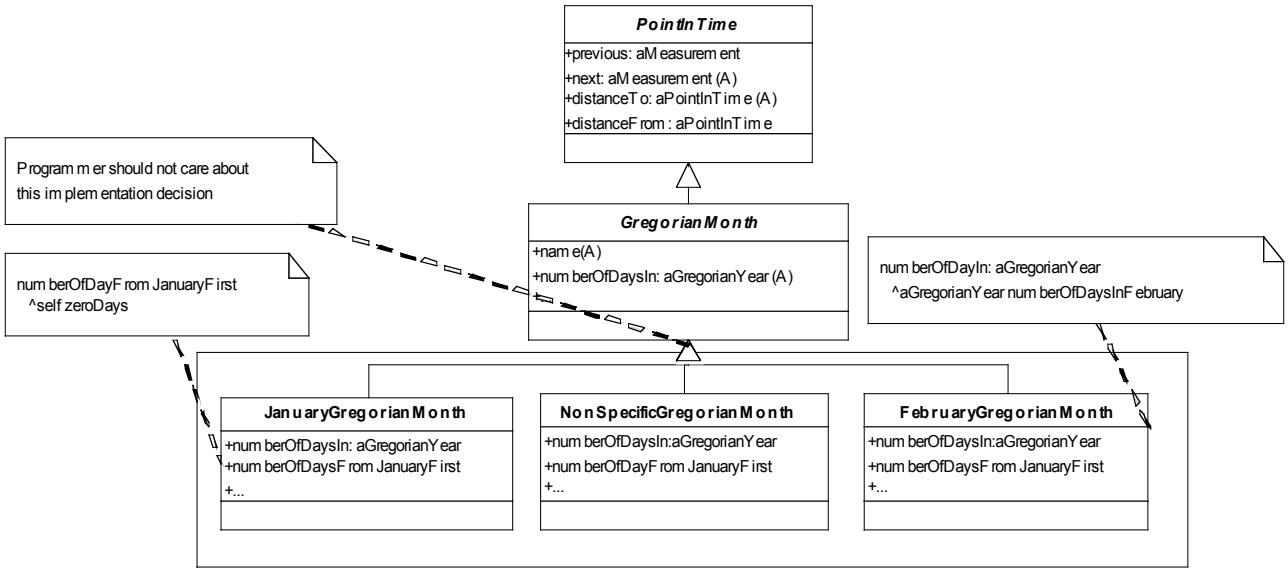

Figure 25: GregorianMonth class hierarchy diagram

When a **GregorianMonthOfYear** receives the message *#numberOfDays,* it only needs to send the message *#numberOfDaysIn:* to its *month* with the year referenced by its instance variable named *year* as parameter of the message. Implementing the irregularity of the Gregorian calendar with specific abstractions for the special cases allowed us to minimize the use of the conditional message *#ifTrue:* to just one place, the creation of a year. Figure 26,27 and 28 show how the objects interact to respond the message *#numberOfDays* when it is send to July 2005, February 2004 (a leap year) and February 2005 (a non-leap year).

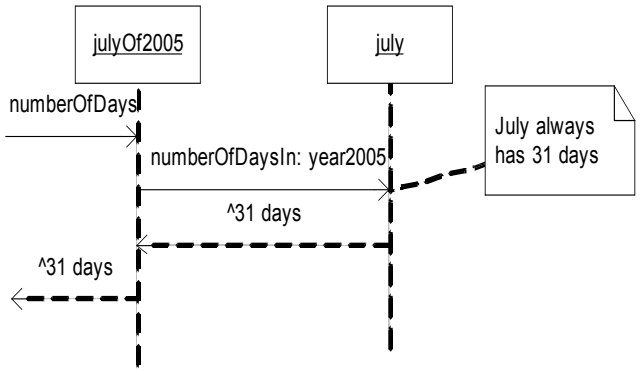

Figure 26: Getting the number of days of a non specific Gregorian month

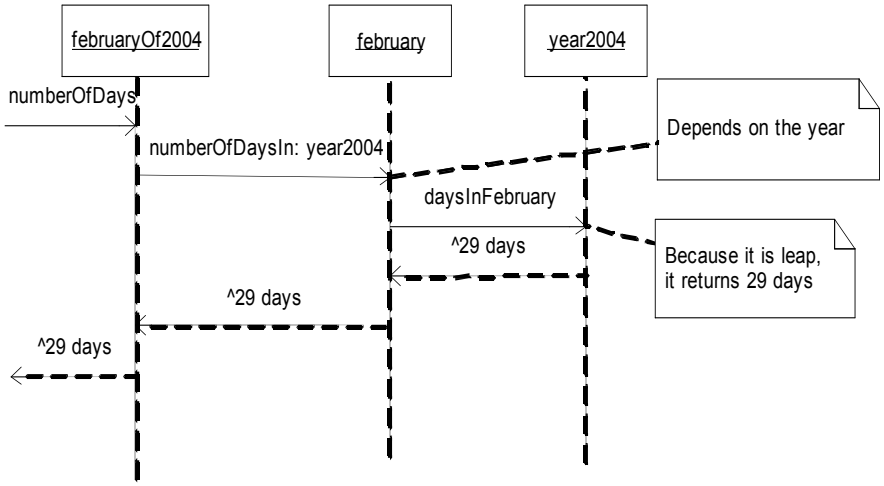

Figure 27: Getting the February's number of days of a leap year

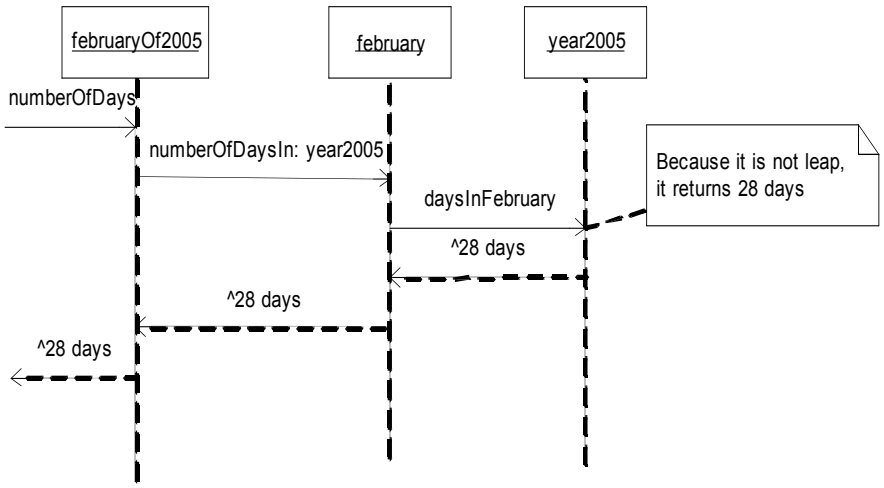

Figure 28: Getting the February's number of days of a non leap year

5.4 Dates

Dates are modelled with the **GregorianDateBehavior** abstract class, that implements common messages for all dates, no matter if they are absolute or relative. **GregorianDate** represents absolute dates and **RelativeGregorianDate** represents relative dates in a time line view with certain time span. The implementation of *#next:aMeasurement* differs on each class. The **GregorianDate** class implements this message moving through the dates of the continuous time line, but the **RelativeGregorianDate** class uses its calendar (an instance of **TimelineView**) to obtain the dates it has to jump through when moving. The message *#distanceTo:aGregorianDate* is implemented in **GregorianDateBehavior** because it can be shared by its subclasses. See Figure 29.

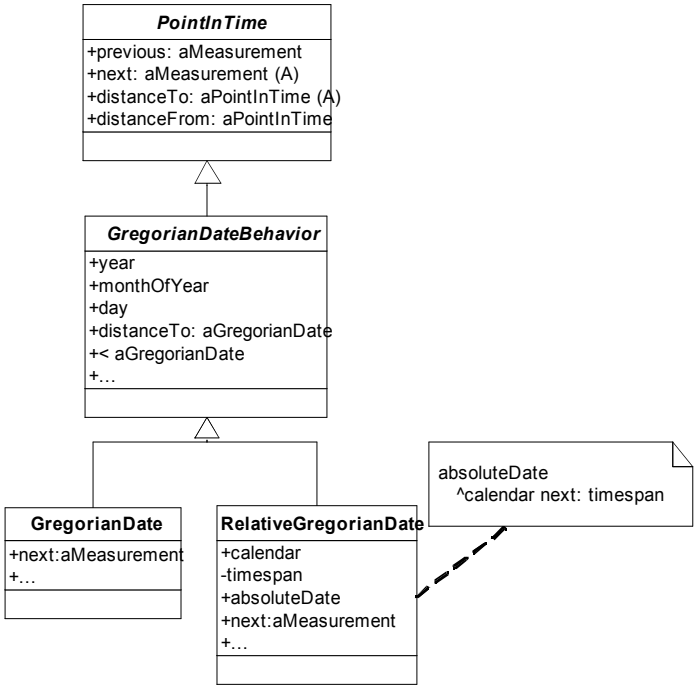

Figure 29: GregorianDate class hierarchy diagram

Figure 30 shows an object diagram of a **RelativeGregorianDate** that represents 10 working days from today, with today equals to July 18th of 2005.

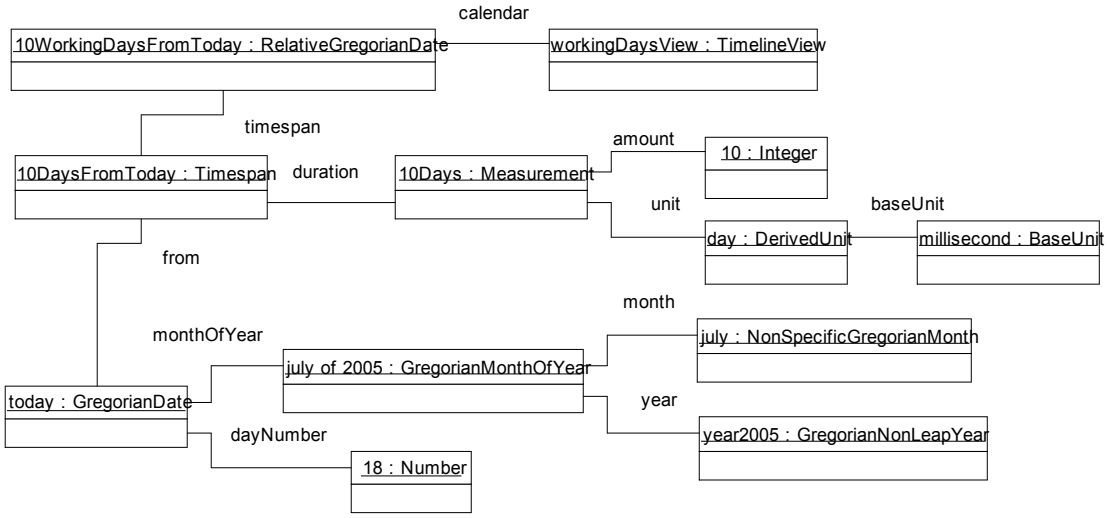

Figure 30: A RelativeGregorianDate object diagram

5.5 Other time entities

A **GregorianDateTime** is composed by a date (instance of **GregorianDate** or **RelativeGreogrianDate**) and a time (instance of **TimeOfDay**). Because the date can be relative, the model also supports relative date times.

The class **TimeOfDay** is implemented with an instance variable that represents the time passed since hour 0, that is a time measurement. That time measurement can be of any resolution (hour, minute, second, millisecond, nanosecond, etc.). If a better resolution that nanosecond is needed, a new time unit can be created with the new resolution to specify more accurate time of days.

The **GregorianDay**, **GregorianDayOfMonth** and **GregorianMonthOfYear** classes are also subclasses of **PointInTime**, but their time line is more a circle than a line. Therefore, the message *#next* returns January 1st when it is sent to December 31st, and the message *#previous* returns December 31st when it is sent to January 1st. Figure 31 shows the class diagram for these time entities.

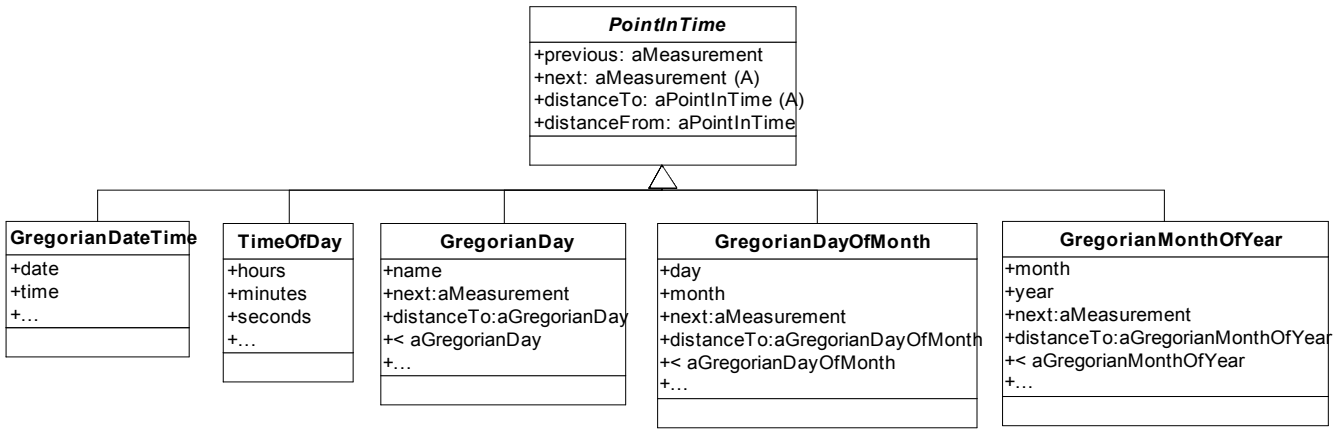

Figure 31: Other points in time class hierarchy diagram

5.6 Timeline segments, Intervals and Timeline Views

The model provides new abstractions which behave like **Collection**. They are **SetDefinedByRules** and **MeasurementInterval**. The former allows the creation of sets where its elements are not added one by one. Elements belong to this set if there is a **SetRule** that returns *true* when the message *#includes*: is sent to it.

MeasurementInterval is provided by the measurement model. It was necessary to create such an abstraction because the Smalltalk class **Interval** can not be used with objects that are not **Number**. It works with any class that defines a total order on its instances, like **Measurement**, **GregorianYear**, **GregorianDate**, etc. Figure 32 shows the class diagram for these classes.

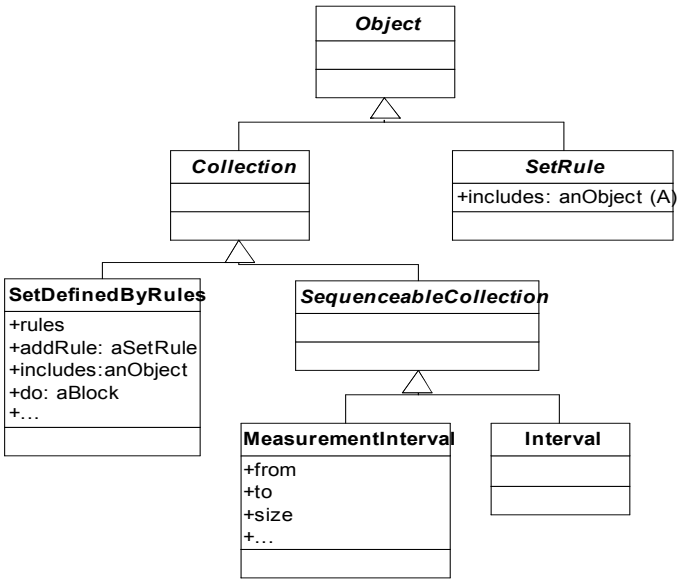

Figure 32: Extension to the Collection class hierarchy

Different subclasses of **SetRule** are provided such as **SpecificObjectSetRule** (used to define a specific object as part of the set), **TransformationSetRule** (used to decorate other SetRule with a transformation block) and **IntervalConstrainedSetRule** (used to filter other **SetRule** to the elements that are part of the interval) among others.

Time line views are reified by three classes: **TimelineViewBehavior,** an abstract class and superclass of **TimelineView** and **NegatedTimelineView**. A **TimelineView** is defined with a **SetDefinedByRules** and the message *#negated* returns an instance of **NegatedTimelineView**. A **NegatedTimelineView** has a **TimelineView** as source. When instances of this class receive the message *#includes:*, it forwards the message to its source and sends the message *#not* to the returned object (a **Boolean**).

Timespan is the class used to represent segments of the time line. It can be used with any **PointInTime** as the starting point (*from*). The *duration* can be any measurement of time. Figure 33 shows the class diagram of these abstractions.

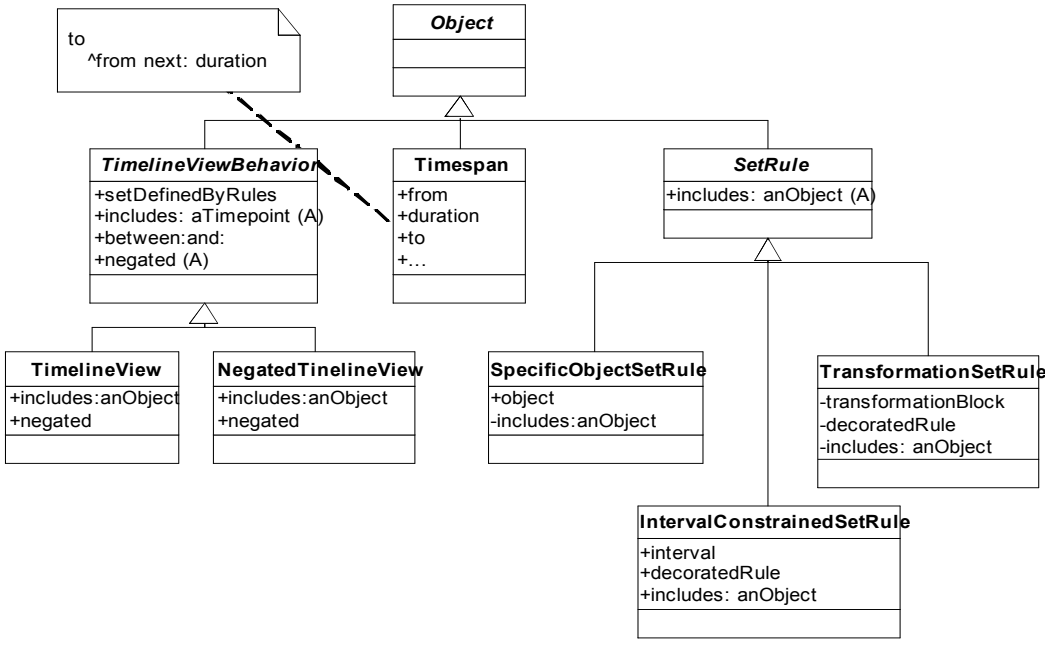

Figure 33: Time filtering and time span class hierarchy diagram

6. RELATED WORK

6.1 Comparison with Smalltalk-80 and Squeak

Figure 34 provides a brief comparison among time abstractions in Smalltalk-80, Squeak and our model. Note that the presented model reifies eleven time entities more than Smalltalk-80 and eight time entities more than Squeak. We present now some concrete examples that show the advantage of having those additional objects, thus proving, oncemore, the importance of reifying as many problem domain concepts (ie.: "model the real world")

6.1.1 Selecting all Mondays of the current year

Figure 35 shows how to obtain all Mondays using the Smalltalk-80 model. First, a collection with the correct number of days of the year 2005 is created. Note that the number 2005 is used to refer to the year 2005 since no special object exists for it (ie.: lack of reification). This collection includes the numbers 1, 2, 3,..., 365 because year 2005 is not leap. A collection containing dates of year 2005 is created using the former collection. Note that the message *#newDay: year:* expects the number of days since January 1st plus one to create the right date, information that most of the people does not know (Does anybody know how many days are between January 1st and July 2nd?). Finally, all Mondays of year 2005 are selected comparing the date's day name with the symbol *#Monday*.

```
yearDayCount := 1 to: (Date daysInYear: 2005).
currentYearDates := yearDayCount collect: [:aDayCount|Date newDay: aDayCount year: 2005 ].
currentYearDates select: [ :aDate | aDate dayName = #Monday ].
```

Figure 35: Smalltalk-80's model example

Figure 36 shows the same problem solved with Squeak's model. With Squeak it is easier to obtain all year's dates but its model still lacks an object to represent a day, therefore, a **Symbol** is needed to compare the day name which is error prone. If Monday is not correctly typed (i.e. *#monday* instead of *#Monday*) the programmer will not get any indication of error and the program will not behave as expected.

```
Year current dates select: [ :aDate | aDate dayOfWeekName = #Monday ].
```

Figure 36: Squeak's model example

With our model, getting the dates of a year is similar to Squeak's model, but because days are reified the message *#isMonday* is sent to the date. An error will be signalled if the message is not correctly typed or if the date protocol changes. See Figure 37.

```
GregorianYear current dates select: [ :aDate | aDate isMonday ].
```

Figure 37: New model's example of use

	Smalltalk-80 and ANSI Smalltalk	Squeak's Chronology package	Presented Model
Year (i.e. 2005)	Modelled as a **Number**	Reified with class **Year**	Reified with class **GregorianYear**
Month of a Year (i.e. January 2005)	Not modelled	Reified with class **Month**	Reified with class **GregorianMonthOfYear**
Date (i.e. 01/01/2005)	Reified with class **Date**	Reified with class **Date**	Reified with class **GregorianDate**
Date and Time (i.e. 01/01/2005 10:00:00 AM)	Reified with class **DateAndTime**	Reified with class **DateAndTime**	Reified with class **GregorianDateTime**
Month (i.e. January)	Modelled as **Symbol**	Modelled as a **Symbol**	Reified with class **GregorianMonth**
Day of Month (i.e. January First)	Not modelled	Not modelled	Reified with class **GregorianDayOfMonth**
Week (i.e. First week of 2005 or Second week of January 2005)	Not modelled	Not modelled	Not modelled
Day (i.e. Monday)	Modelled as **Number** and **Symbol**	Modelled as **Number** and **Symbol**	Reified with class **GregorianDay**
Time (i.e. at Noon, 10:00:00 AM)	Reified with class **Time**	Reified with class **Time**	Reified with class **TimeOfDay**
Time distance (i.e., 1 year, 3 months, 10 days, etc.)	Reified with class **Duration**. Expressed only in terms of seconds	Reified with class **Duration**. A duration of 1 month is converted to 31 days	Reified as **Measurement** with **Units** such as: year, month, week, day, hour, minute, second, millisecond, decade, century, millennium or any other time unit.
Time line segment (i.e. From 01/01/2005 with a length of 10 days)	Not modelled	Reified with class **Timespan**, with a start and a duration	Reified with class **Timespan**, with a start and a distance expressed as measurement
Time line interval with different granularity (i.e. From 01/01/2005 to 01/02/2005, or from January 2005 to July 2005 every 2 months)	Not modelled	Not modelled	Reified with class **MeasurementInterval** with a measurement as step. Also know as time point occurrences
Relative Dates (i.e. 10 working days from 01/01/2005)	Not modelled	Not modelled	Reified with class **RelativeGregorianDate**
Time line views	Not modelled	Not modelled	Reified with class **TimelineView**
The end of time	Not modelled	Not modelled	Reified with the object *theEndOfTime*
The beginning of time	Not modelled	Not modelled	Reified with the object *theBeginningOfTime*

Figure 34: Comparing Smalltalk-80, Squeak and the presented time model

6.1.2 Getting the last dates of every month of a year

Figure 38 shows how to solve this problem with the Smalltalk-80 model. We can observe the same issues as in the previous example because all the dates of the year have to be created and because a month of a year is not reified the message *#daysLeftInMonth* is sent to a date.

```
yearDayCount := 1 to: (Date daysInYear: 2005).
currentYearDates := yearDayCount collect: [:aDayCount|Date newDay: aDayCount year: 2005 ].
currentYearDates select: [ :aDate | aDate daysLeftInMonth = 0 ].
```

Figure 38: Smalltalk-80's model example

Figure 39 shows ours model solution. Because months of a year are reified, a collect on each month of a year is performed sending the *#lastDate* message to each of them. This solution has a better performance than the Smalltalk-80 one because the collect is done over

twelve elements (the twelve months of a year) while the Smalltalk-80 does a select over 365 dates. The Squeak solution is similar to the presented model.

```
GregorianYear current months collect: [ :aMonthOfYear | aMonthOfYear lastDate ].
```

Figure 39: Squeak's model example

6.1.3 Obtaining the number of months between two months

Figure 40 shows the problem solved with the Smalltalk-80's model. Since the Smalltalk-80's model does not deal with "months of a year", a mathematical expression has to be programmed to solve the problem every time we need to do so. In the code, we show there is no verification about the month number or year number, thus they could be invalid. This is a very common mistake that leads to invalid behaviour. This piece of code should be encapsulated to avoid mistakes and code duplication. Note also that the result of that expression is a number.

```
fromMonthNumber := 6.
fromYearNumber := 2005.
toMonthNumber := 12.
toYearNumber := 2010.
numberOfMonths := 12 – fromMonthNumber + (toYearNumber – 1 – fromYearNumber * 12) + toMonthNumber
"It returns the number 66"
```

Figure 40: Getting the number of months between two months with Smalltalk-80's model

Figure 41 shows the same problem solved with Squeak. The Squeak model allows the programmer to deal with months (an abstraction we call **GregorianMonthOfYear**) but a **Timespan** has to be created to obtain all the months, and then the size of that segment is used to get the final result. Note that it also returns a number.

```
((Month month: 6 year: 2005) to: (Month month: 12 year: 2010)) months size
"Returns the number 66"
```

Figure 41: Getting the number of months between two months with Squeak's model

Figure 42 shows our model's solution. Because "month of a year" is reified, the *#distanceTo:* message is sent to the first one with the second one as a parameter. Note that the returned object is not the number 66 but a measurement of time; in this case, measured in months: the object *66 months*.

```
'06/2005' asGregorianMonthOfYear distanceTo:  '12/2010' asGregorianMonthOfYear
"Returns 66 months, not just 66"
```

Figure 42: Getting the number of months between two months

6.2 Comparison with Chronology Squeak's package

The main difference between our model and Squeak's one is how time entities are understood. In our model, time entities are points in the time line and measurements are used to represent time distances. In Squeak, time entities are segments in the time line modelled with the class **Timespan**. For example, **Month** is a subclass of **Timespan**, so it behaves like a time segment. Therefore, the object created with the expression "*Month month: 13 year: 2010*" (note that a month number thirteen is invalid in the Gregorian calendar) is the same as "*Month month: 1 year: 2011*", because they are the same segment.

Because **Timespan** is the superclass of all time entities in Squeak, it has confusing protocol such as *#lastDate* or *#firstDate* and strange behaviour when comparing time entities. Messages such as *#lastDate* and *#firstDate* make sense when they are sent to a year or a month of a year, but they loose meaning when the receiver is a date or a date time.

Squeak allows comparing points of different resolution because time entities are modelled as segments, as it is shown in Figure 43.

```
"Returns true if today is not the first day of the current year"
Year current < Date today
"Returns true if today is not the first day of the current moth"
Month current < Date today
```

Figure 43: Comparison in Squeak

We believe this is confusing and inconsistent with the analogy of time entities with segments. It is confusing because it does not make sense to ask "*Is the current year before today?*" How can a year be compared with a date? Only if a year is seen as a segment from the beginning of time to the first day of that year this question can be answered. But that is not the "common" meaning of year. A year is not the first day of that year.

Likewise, Squeak does not model a year as a segment starting at year 1 with a duration of the passed years; it models a year as a segment that starts at hour 00:00:00 of January 1st of that year, with a duration of 365 or 366 days. Therefore, we thought the #< message meant "does the segment receiving the #< message **include** the one given as parameter?", but that is not the behaviour of #<. It does not mean

"does the segment receiving the #< message **intersects** the one passed as parameter?" either. We could not find a consistent meaning for the #< message when sent to these objects.

Figure 44 shows the peculiar behaviour of the #< message. The behaviour when comparing a year with the first month or date of that year is the most puzzling one. They are not less, nor greater or equal between them. Strange behaviour is observed also when comparing instances of **Timespan** with its subclasses. To avoid this type of confusion our model does not allow points of different resolution to be compared as we showed and demonstrated before.

"All these comparisons return false"

(Year year: 2005) < (Month month: 1 year: 2005).

(Year year: 2005) > (Month month: 1 year: 2005).

(Year year: 2005) = (Month month: 1 year: 2005).

"This collaboration shows 'inclusion' behaviour"

(Year year: 2005) < ((Month month: 2 year: 2005) to: (Month month: 3 year: 2005)).

"These collaboration shows 'intersection' behaviour"

(Year year: 2005) < ((Month month: 2 year: 2005) to: (Month month: 3 year: 2006)).

((Month month: 12 year: 2004) to: (Month month: 2 year: 2005)) < (Year year: 2005).

Figure 44: Puzzling behaviour of #< message in Squeak

6.3 Comparison with other research work

Barbic and Pernici [4], in their work related to office information systems, propose a similar model to ours. The concept of time they present is based on a discrete temporal axis (our notion of time line) where time points can be mapped to integers, but it differs from our work because they provide a quantum of minute to every point, while in our work time entities are part of different time lines, each one with its own quantum. They also mention that "time is infinite in the past and in the future", but it is not clear how they represent that characteristic in the final design, while our model provides two objects to reify those entities, *theEndOfTime* and *theBeginningOfTime*. Their model supports absolute and relative time entities, but they do not provide abstractions to filter the time line, so relative time entities fall in what we model as time spans. They propose to return "UNKNOWN" when comparing time points of different granularity which differ from our solution where such comparisons are not allowed, but they do not allow having specifications that cannot be converted to a same level of granularity. Week days, days of month and months are not modelled as first class entities.

Goralwalla et al [13], in their work related to database management systems, propose a model that introduces the idea of temporal granularity, which is "a special kind of unanchored temporal primitive that can be used as a unit of time". Such entities are related to the time units we propose in our model. The anchored time entities are what we represent as point in the time line, and unanchored entities are what we model as time measurements. Due to the Gregorian calendar irregularity, they propose to use indeterminate spans to represent conversions between measurements of not related units. For example, *1 month* would be converted to *[28 days – 31 days]* therefore, comparing *1 month* with *30 days* would return UNKNOWN, the same solution adopted by [4].

To represent anchored times they use the concept of time granule defined by Bettini [9], where an anchored entity is "an interval on the global timeline" which differs from our work where anchored entities are modelled as points in time. Because they use intervals they have to differentiate three types of interval, *Beginning Instant*, *Determinate Interval* and *Indeterminate Instant*. Examples of *Beginning Instant* are 1995_{beg}, January 1995_{beg} and January 1^{st} of the year 1995_{beg} , that return true when compared for equality. Our approach is simpler because there is no need to implement different types of interval and we treat years, months of years and dates as completely different time entities, therefore, no confusion about the year 1995 and the month of year January of 1995 is allowed.

When moving from a point a measurement of time with higher resolution that the quantum of the point, for example 3 days from January of 1995, they assume they are moving from the first point of the same granularity contained in the former point, which means 3 days from January 1^{st} of 1995 for the presented example. We see this as an arbitrary solution, therefore, we do not allow this type of expressions.

The ODMG time model [10] is similar to the Smalltalk-80 one; therefore, it lacks important abstractions and has the same problems mentioned in this paper as Smalltalk-80, such as representing years and months with numbers, modelling time distances with numbers and so on. Bertino et al, [8] proposes an extension to the ODMG model providing temporal information to objects and a new abstraction, time interval. Although this model helps to keep historic information about objects, abstractions to represent time spans, relative time entities, time granularity among others are left as future work. Huang et al [15] also extends the ODMG model but adding a new dimension to the temporal one: the spatial dimension. Although this new dimension is being taken into account to keep historical information about objects, the time model has the same limitations as [8].

Goralwalla et al [14] presents in a newer paper an object-oriented framework that provides a unified infrastructure to support temporal entities. The framework divides the time domain in four dimensions: Temporal Structure, Temporal Representation, Temporal Order and Temporal History. The Temporal Structure classifies temporal entities as anchored and unanchored. Anchored time entities are classified as instants and intervals. Both, anchored and unanchored, can be discrete or continuous and determinate or indeterminate. Our model supports this classification but some of the entities we provide do not behave as they proposed in [13] like it is previously explained. The Temporal Representation dimension is automatically provided by the classes composing our model. The Temporal Order is also provided in our model by means of protocol that each time objects respond to which, at the same time, are polymorphic no matter the type of the object. The Temporal History can be achieved using the collection objects and the time objects provided in our model.

The Java time model is completely different from our's. The Java Language [16] provides a single abstraction named **Calendar** to handle all types of time entities. **Calendar** is an abstract class that has concrete subclasses such as **GregorianCalendar**. This class is a combination of fields that are set with the message *set (int field, int value)* and get with the message *get(int field)*, being *field* a number that represents the time entity to be changed. For example: *set (Calendar.MONTH, Calendar.JANUARY)* changes the month of that calendar instance to be January.

An instance of **Calendar** with just the field *Calendar.YEAR* represents a year, an instance of **Calendar** with the fields *Calendar.YEAR* and *Calendar.MONTH* represents a month of a year and so on. Because **Calendar** represents all types of time entities, no specific protocol is provided to objects that represent specific time entities such as years, months, days, etc. For example, there is no message such as *#dates* to obtain all dates of a specific year. On the contrary, it provides generic protocol like *#isLeap* that can be answered by any instance of **GregorianCalendar**, such as dates. This ambiguity makes the model confusing, difficult to learn and use. For instance, the distance between two "calendars" is represented by the number of milliseconds that separate those "calendars". Therefore, if a year is compared with a date, the number of milliseconds since January 1st of that year to the hour 00:00:00 of the compared date is returned.

We believe this model suffers from the same design issues that any generic model. Real-world concepts should have a one-to-one mapping with the classes provided by object models. The Java model does not follow this rule, it joins the concept of year, month of year, date and date time in one concept they call **Calendar**; therefore, it provides a one-to-many mapping between real world concepts and model concepts, creating a different language that the one used in the problem domain. Note that when comparing time entities we use the word "calendars" which is completely unreal, we do not compare calendars, we compare years, months, dates, etc.

This model also lacks abstraction to represent time measurements, time intervals, time spans, days of months, relative dates and time line filters which are important entities of the time domain. This lack of abstractions produces the programmer to create their own implementation of such as entities.

The .NET model [20] is similar to the Java one. It provides an abstract class named **Calendar,** which is subclassed by **GregorianCalendar**. In contrast to the Java model, the messages such as *AddDays*, *AddHours* sent to a **Calendar** to move through the time line do not return instances of **Calendar** but instances of **DateTime**. A **DateTime** is a "Structure" that measures the number of 100-nanoseconds since a particular origin defined by the calendar they belong to, for example January 1st of year 1 for the Gregorian calendar. An abstraction called **TimeSpan** is provided to represent time measurements expressed in 100 of nanoseconds. No different time units are provided. The .NET model has the same modelling problems as the Java one. It lacks important abstractions and it has the same design flaws.

7. CONCLUSION

This paper presents an object model that focus on the representation of the Gregorian calendar time entities. It is based on a simple metaphor where time entities are represented by points in the time line. Those points have different resolutions and they include points of higher resolution.

The model provides a total order between time points which allows programmers to determine which point comes before or after another, go from one point to another and obtain the distance between two points.

Because time entities are analogous to points within a line, the model permits the representation of segments of the time line and it provides abstractions to create intervals between points.

A distinguishing feature of this model is that it uses a generic Measurement Model to reify Time Measurements. This modelling decision allows programmers to share the concept of measurements of time with any other type of measurement and it permits to operate arithmetically with them.

Time line views created to filter time line points is another important feature. Relative points in time can be created based on these views.

The model also provides abstractions for time entities such as a day, a day of a month and months.

7.1 Concrete Benefits

The main benefit obtained with this model is that complex observations of the time domain can be easily programmed. Although this characteristic is difficult to be formally proved, it can be inferred because of the provided abstractions and protocol. This model is being used as "the" time model at Mercap Inc. in all its new applications. It proved us to be very powerful and easy to use.

7.2 Lessons Learned

7.2.1 Create only valid objects

Objects should only exist if they are valid. For example, a Gregorian year can only exist if its number is not zero. This rule, combined with the rule of immutable objects, gives programmers the security that the objects they work with are always valid.

This rule also implies that invalid or incomplete objects should be represented with specific abstractions. Builders [28] are an example of this type of objects. When a builder is created it is "incomplete" because it can not build the desired object until all the information of that object is provided. The builder will be modified until it reaches a state where it is able to create the specified object.

7.2.2 Immutable Objects

The implementation of time entities as immutable objects simplified the model's design and implementation. Not only they provide the benefits mentioned in [5], but they also avoid non-contemplated consistency problems that could appear during an object's life cycle.

Immutable objects that are valid from the time they are created ensure the programmer that she's not dealing with invalid ones, because an object is not instantiated if its preconditions are not met.

7.2.3 Development Technique

We cannot conclude this paper without mentioning the advantages we obtained due to the use of the "Test Driven Development" technique. (See [7] and [6]). Each observation we made of the time domain was programmed as a test that we took as the starting point to implement and improve the model.

It is also necessary to highlight the advantages that a dynamically typed and late binding programming language offers when using this technique. It is because of the dynamically typed characteristic of Smalltalk that we could make our model evolve smoothly. The late binding characteristic allowed us to "program on demand" completely within the debugger, defining classes, methods and instance variables as required by the tests, a characteristic still very restricted in languages such as Java or C#.

7.3 Future work

We need to research the addition of time zone entities in our model. The time zone adds some complexity because we would like date times such as January 1st of 2005 at 10:00:00 in Buenos Aires, Argentina to be equal to January 1st of 2005 at 11:00:00 in Montevideo, Uruguay

The **Timespan** protocol is limited at this time. We need to expand it with protocol related to line segments.

New abstractions need to be created like **Hour**, **Minute**, etc. We have not created them yet because measurements are used to represent these entities. One advantage of having a class to represent hours is that an hour less than 0 or greater than 23 could not be created.

We have not reified time lines. We think that modelling time lines would simplify the implementation of moving along them or along lines of different resolution.

At this moment the model implements relative dates as the only relative points, but there is no reason to have such a limitation. We will expand the model to support any point in time to be relative.

Mercap Inc, is studing the open source licences to open this model to the Smalltalk community.

8. ACKNOWLEDGMENTS

We would like to thank Mercap Inc.'s Software Development Team, for their comments and use of the Time model. Also, we would like to thank Michael Maximilien of the IBM Almaden Research Center for his review and friendship.

9. REFERENCES

[1] Allen, E., Chase, D., Luchangco, V., Maessen, J. and Steele, G. *Object-Oriented Units of Measurement*. Technical Report, OOPSLA 2004

[2] Allen, F. *Maintaining Knowledge about Temporal Intervals*, Communications of the ACM, November 1983, Volume 26, Number 11

[3] ANSI Smalltalk - http://www.smalltalk.org/versions/ANSIStandardSmalltalk.html

[4] Barbic, F., Pernici, B. *Time modeling in Office Information Systems*, ACM SIGMOD International Conference on Management of data, Proceedings,1985

[5] Baümer, D., Riehle, D., Siberski, W., Lilienthal, C., Mergert, D.,Sylla, K. and Züllighoven, H. *Values In Objects Systems*. UBILAB Technical Report, 1998-10-10, Zurich, Switzerland

[6] Beck, K. *Test Driven Development: By Example*. Addison-Wesley, Reading, MA, 2002

[7] Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999

[8] Bertino,E., Ferrari, E., Guerrini, F., Merlo, I. *Extending the ODMG Object Model with Time*,ECOOP'98, SPringer-Verlag Berlin Heidelberg 1998

[9] Bettini, C, Dyreson, C.E., Evans, W.S., Snodgrass, R.R., Wang, X.S. *A Glosary of Time Granularity Concepts*, in *Temporal Databases – Research and Practice*, 1998

[10] Cattel, R. *The Object Database Standard: ODMG93*, Morgan-Kaufmann,1996

[11] Corsetti, E., Montanari A., Ratto, E. *Dealing with Different Time Granularities in Formal Specications of Real-Time Systems*. The Journal of Real-Time Systems, 1991.

[12] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[13] Goralwalla, I., Leontief, Y., Özsu, M., Szafron, D. *Temporal Granularity: Completing the Puzzle*, Kluwer Academic Publishers, Boston

[14] Goralwalla,I., Tamer Özsu, M, Szafron, D. *A Framework for Temporal Data Models: Exploiting Object-Oriented Techology*, Proceedings of TOOLS'97, IEEE.

[15] Huang, B., Claramunt, C. *STOQL: An ODMG-Based Spatio-Temporal Object Model and Query Language*, Symposium on Geospatial Theory, Preocessing and Applications, Ottawa 2002

[16] http://www.javasoft.com

[17] Kennedy, Andrew J. *Programming Languages and Dimensions*. PhD Thesis, University of Cambridge. Published as

Technical Report No. 391, University of Cambridge Computer Laboratory, April 1996

[18] Maiocchi, R., Pernici, B., Barbic, F. *Automatic Deduction of Temporal Information*, ACM Transactions on Database Systems, Vol. 17, No. 4, December 1992

[19] Montanari, A., Maim, E., Ciapessoni, E., Ratto, E. *Dealing with Time Granularity in Event Calculus*. International Conference on Fifth Generation Computer Systems, Proceedings, Tokyo, 1992.

[20] http: //www.microsoft.com

[21] Pinkeny B., *Squeak Chronology Package*, http://minnow.cc.gatech.edu/squeak/1871

[22] Reingold, E., Dershowitz, N. *Calendrical Calculations: The Millennium Edition*. Cambridge University Press, Reading, 2001

[23] http://www.squeak.org

[24] Wang, X.S., Jajodia, S., Subrahmanian, V.. *Temporal Modules: An Approach Toward Temporal Database*s. ACM SIGMOD International. Conference on Management of Data, Proceedings, 1993.

[25] Wilkinson, H., Prieto, M., Romeo, L. *Arithmetic with Measurements on Dynamically-Typed Object-Oriented Languages*, OOPSLA 2005.

[26] Maiocchi, R. Pernici,B., *Temporal data management Systems: A compartive view*. IEEE Trans. Knowl. Data Eng. December 1991.

[27] Maiocchi, R. Pernici,B., Barbic, F. *Automatic Deduction of Temporal Infromation*. ACM Transactions on Database Systems, December 1992.

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

Microprints: A Pixel-based Semantically Rich Visualization of Methods

Romain Robbes

Universitá della Svizzerra Italiana, Faculty of Informatics, Lugano, Switzerland

Stéphane Ducasse

University of Bern, Software Composition Group, Switzerland
Université de Savoie, Language and Software Evolution Group-LISTIC, France

Michele Lanza

Universitá della Svizzerra Italiana, Faculty of Informatics, Lugano, Switzerland

Abstract

Understanding classes and methods is a key activity in object-oriented programming, since classes represent the primary abstractions from which applications are built, while methods contain the actual program logic. The main problem of this task is to quickly grasp the purpose and inner structure of a class. To achieve this goal, one must be able to overview multiple methods at once. In this paper, we present *microprints*, pixel-based representations of methods enriched with semantical information. We present three specialized microprints each dealing with a specific aspect we want to understand of methods: (1) state access, (2) control flow, and (3) invocation relationship. We present the microprints in conjunction with the class blueprints of the CODECRAWLER visualization tool [12] and also integrated into the default code browser of the Smalltalk VisualWorks development environment.

Key words: Object-Oriented Programming, Program Comprehension, Visualization

Email addresses: romain.robbes@unisi.ch (Romain Robbes), stephane.ducasse@univ-savoie.fr (Stéphane Ducasse), michele.lanza@unisi.ch (Michele Lanza).

1 Introduction

In object-oriented applications, classes describe the state of objects and define their behavior. However, objects being behavioral entities, understanding methods is crucial for the comprehension of object-oriented applications [22]. In addition to traditional control flow analysis, there is a large variety of information that can be used to understand a method: how the state of an object is accessed, if and how ancestor state is used, how an object uses its own methods or the methods defined in its superclasses [5], and how an object communicates with other objects.

This topic has already been partly addressed by prior work. Cross et al.defined and validated the effectiveness of Control Structure Diagrams (CSD) [3] [7] which depict the control-structure and module-level organization of a program. Even though CSDs are applied to Ada and Java code, they do not support OOP concepts such as inheritance, overridden methods . . . , but only control flow constructs. SeeSoft [6] can visualize large amount of code but it associates a color to a complete line of code and does not introduce a specific visualization for method semantics. Such a visualization is commonly used in aspect-browser tools. However, it does not provide object-oriented specific information either. Jerding and Stasko [9] proposed to use a mural visualization to represent program execution but does not propose specific object-oriented method level visualizations. sv3D, developed by Marcus et al., presents lines of code as dots and each dot can be associated with different information such as the nesting level or the control flow [13]. For quantitative information, such as the occurrence of a phenomena, 3D is used. However, sv3D is more a general visualization approach than a fine-grained one specialized to convey important aspects of object-oriented code.

Our approach is based on *microprints*, pixel-based character-to-pixel representations of methods enriched with semantical information mapped on nominal colors.

The paper is structured as follows: first, we highlight the key constraints of the work presented. Then we present microprints and the three instances we defined. The next section shows how microprints are integrated with the VisualWorks Smalltalk development environment and how they enhance the class blueprint visualizations in CODECRAWLER [12]. We conclude with a discussion and a comparison of our approach with related works.

2 Constraints

When working on method understanding and visualization we have to consider the following constraints:

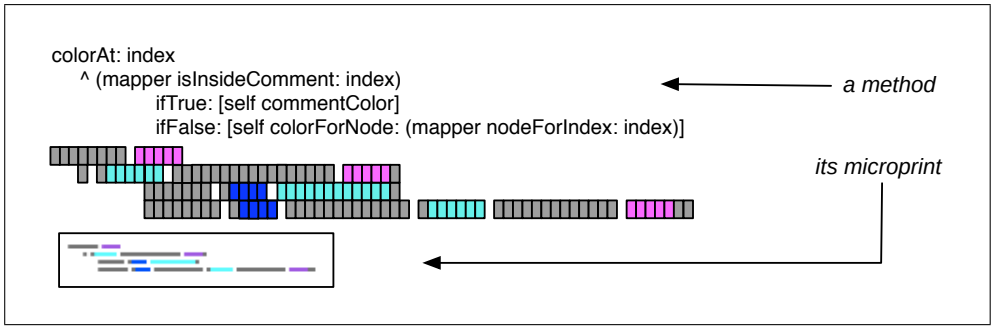

Fig. 1. The principle of a microprint.

Context switches. We want to avoid these as much as possible as they induce latency: The human brain is much faster at glancing at information than at restoring contexts.

Limited space. Screens are still too small and as extra information should not clutter the code, it is crucial that visualizations can be effective in a limited amount of space.

Limited number of colors. As the human brain is not capable of simultaneously processing more than ten distinct colors, a diverse but small number of colors should be used [20] [21].

Pixel aliasing. Pixel juxtaposition produces aliasing. Therefore to get a clear picture (without unintended extra colors) the colors should be well chosen.

Information interpretation. The information should be clear and interpretable at a glance. In particular color conventions have to be consistent.

3 Microprints

A microprint is a character to pixel mapping of a method annotated with semantical elements. Compressing whole words to a single pixel was not done, as one pixel per word would involve some translation duty for the user as the microprint would no longer look similar to the method it visualizes. Figure 1 shows how each character of the method body is represented as a pixel in a microprint. Although Smalltalk is used in examples throughout this paper, Microprints can be applied to any object-oriented language.

We decided to use distinct nominal colors to ease the interpretation of the microprints. In Table 1 we see the color mapping schema we apply throughout this paper [1] . The color mapping is consistently used over the different microprints that we present in the following section. For example, the blue color is used consistently to represent the object itself. In addition, program elements which are not marked in any way by a microprint are colored in gray, whereas comments use a lighter shade of gray.

[1] A black-and-white copy of this paper will be very hard to understand.

Description	Color
Microprint - State Changes and Accesses	
Instance variables	Cyan
Accessor method to an instance variable (read)	Cyan
Local variables and arguments	Purple
Self pseudo-variable (this)	Blue
Super pseudo-variable	Orange
Reference to a class or global variable	Yellow
Assignment operator	Red
Accessor method to an instance variable (write)	Red
Microprint - Control Flow	
Return	Red
Use of exceptions	Red
Conditional control structures	Blue
Iterating control structures	Green
Blocks of code (varies with nesting level)	Purple
Microprint - Object Interaction	
Message to self	Blue
Message to super	Orange
Message to other	Purple
Message to classes	Yellow

Table 1
The color mappings used for the microprints.

Microprints keep code familiarity by preserving the shape and indentation of the code, as this is an important information for programmers. In addition, this creates a one-to-one mapping between the code and its representation forms to avoid programmers getting “lost in translation”.

However, problems may occur if this approach is applied naively:

- Important information such as returns or conditionals are sometimes not visible

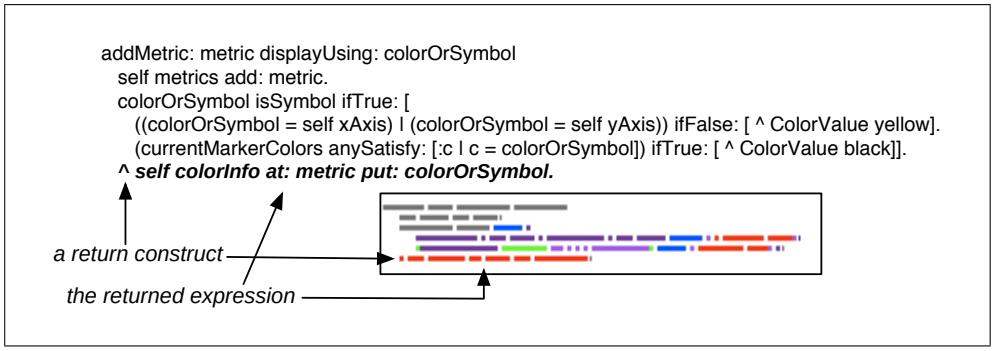

Fig. 2. Propagation of colors from program elements

enough. For example, in Smalltalk, method returns are expressed using the caret character ^ and not with a keyword such as `return`.

- When the code is composed of nested structures such as nested conditionals and loops, identifying the scope of a given structure is crucial. Representing characters directly does not provide enough visual feedback and produces aliasing effects.

To solve these problems the mapping of the color is not direct but propagated to the nested elements. In Figure 2, the entire expression returned (last line of the method) is also colored in red. Each new nesting element however takes precedence over the color of its parent: a return expression contained in a conditional one will not have the blue color of the conditional expression but the red of the return expression, as shown by the end of the lines 4 and 5 in Figure 2. This solution does not adress the problem of the identification of the scope of a construct but provides a better visual feedback.

4 Dedicated MicroPrints

When reading object-oriented code, the key information that the programmer is looking for can be classified into the following categories : (1) state changes and accesses, (2) method control flow and (3) method invocations or object interactions. Putting all this information into a single microprint would lead to an unreadable picture, since far too much information would be displayed (the same applies for code highlighting). Since for humans it is easier to combine information rather than to extract it, we propose three microprints specialized on each of these aspects. These microprints can be displayed alongside a method body. Since they are significantly smaller than the method itself, we can display at least 3 of them in the same space without having scrolling problems, as shown on the right of Figure 10.

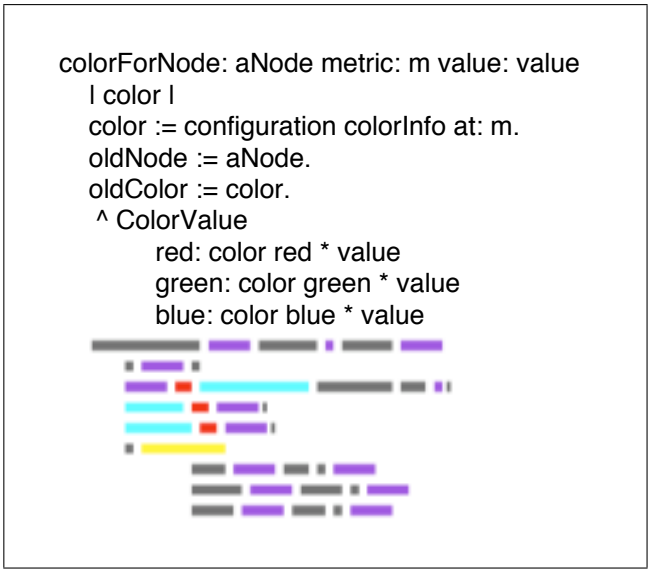

Fig. 3. A visualization of the method colorForNode:metric:value: using a dedicated microprint for state changes and accesses.

4.1 Microprint - State Changes and Accesses

The intention of this microprint is to convey how variables of different scopes are manipulated. This microprint focuses on state accesses and changes. It distinguishes variable scope and assignments.

Color Mapping. Assignments are displayed in red. Different kinds of variables are distinguished: method arguments (purple), the self variable [2] (blue), instance variables (cyan), temporary variables (purple) and global variables such as classes (yellow). The super pseudo-variable is shown in orange as it refers to another class higher in the hierarchy. Some extra analysis is performed to use the same color for accessor methods and direct accesses. Figure 3 presents an example of microprint with state changes and accesses.

Spotting patterns. Glancing at the microprints, one can immediately see some interesting sequences of colors. Cyan-red means that instance variables are set. Purple-red means that local variables are assigned. Yellow spots reveal references to other classes and in general creation of objects of these other classes.

Figure 4 shows two microprints of a lazily initialized accessor method named comboAspect. This method tests if the value of the variable is nil; if this is the case the value is set before being returned. The order of the colors in the microprints allows us to spot this pattern easily. The cyan-red-yellow sequence in the state microprint (a variable is set to an external reference, probably a new instance of the class) and the red-blue sequence in the control flow (returning the result of a conditional

[2] Corresponds to this in Java.

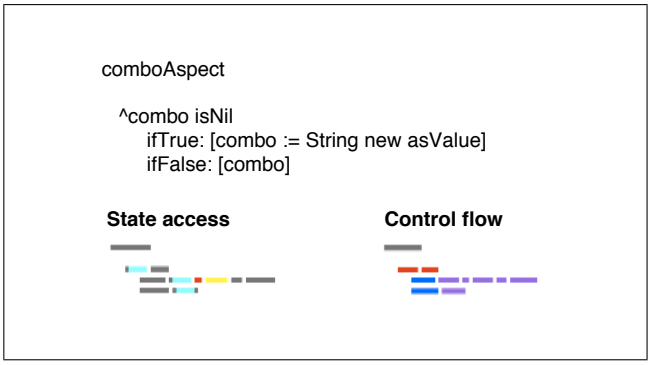

Fig. 4. Microprints of an accessor method following the lazy initialization pattern

expression) is a strong characteristic.

4.2 Microprint - Control Flow

This microprint focuses on method control flow. It highlights the following types of information: loops, conditional statements, conditional loops, return statements, and exceptions.

Color Mapping. Conditional statements are marked as blue, loops as green and exceptions or return statements as red, since they both end the execution of the method. Blocks of code are shown in purple.

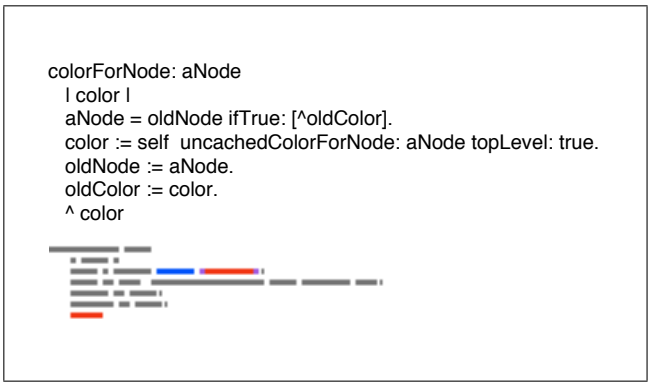

Fig. 5. The control flow microprint of the method colorForNode: reveals it contains a guard clause.

Spotting patterns. Figure 5 shows the microprint of the method colorForNode:. We see there the simple control flow of a method with a guard clause, *i.e.*, one conditional and a return, followed by several statements and a final return statement.

Figure 6 shows a typical control flow microprint of a method with a complex logic. On it we can spot a conditional (blue), conditional loops (green), and explicit control flow returns (red).

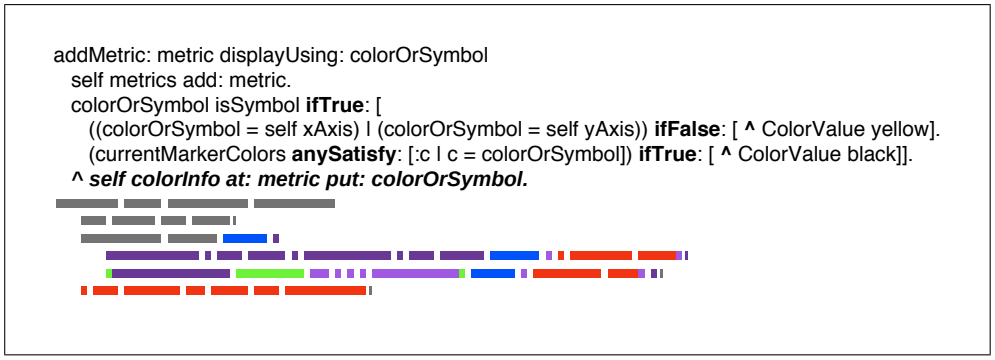

Fig. 6. A complex control flow microprint.

The absence of patterns in a method is another source of information. Such methods do not exhibit any non-linear control flow. This allows one to easily tell apart methods performing some initialization, forwarding messages to other objects, or performing a series of subtasks. Methods with a linear control flow are either totally gray or they only have a single red return spot as their last statement.

4.3 Microprint - Object Interactions

The third dedicated microprint focuses on the different types of method calls, *i.e.*, if a message is sent to another object or is invoked via super or self/this. In such a case, the microprint also indicates whether the method is locally defined or inherited by a superclass.

Color Mapping. Messages sent to self are shown in blue, and messages sent to super, or sent to self but implemented in the superclasses, are displayed in orange. Interactions with other objects are also considered, and are displayed in purple, as we can see on Figure 7. Thus the color choice is consistent with the one used in the state changes and accesses microprints, as shown in Table 1. This consistency allows the user to interpret microprints faster.

Fig. 7. Object interaction microprint. self in blue, super in orange, other in purple

Spotting patterns. This microprint allows one to easily discover the type of interaction a given class has with other classes: whether it is auto-sufficient, relying on its superclass for certain behaviors, or interacts with “foreign classes”. Categoriz-

Fig. 8. A method collaborating only with external objects. Yellow colors message sends to classes, purple message sends to variables.

ing classes or sets of methods in such a way can help the programmer to pick an area of a class which is easier to understand according to his current needs (like understanding the internal implementation of a class, or its relations with its superclass). This microprint also allows one to detect areas where helper methods are used (lots of self or super message sends).

The exceptional cases are also interesting: A method with absolutely no interaction is either an accessor to an instance variable or to a constant. A method with only foreign interactions, such as the one displayed in Figure 8, is really a utility method, and probably never accesses the state of the object. It could come from a previous refactoring.

Fig. 9. An overview of the method protocol “evaluating” of the class RBASTEvaluator, using the state access microprints.

5 Microprints at Work

Microprints have been introduced in the professional IDE of VisualWorks Smalltalk and in CODECRAWLER in the context of class blueprints [12].

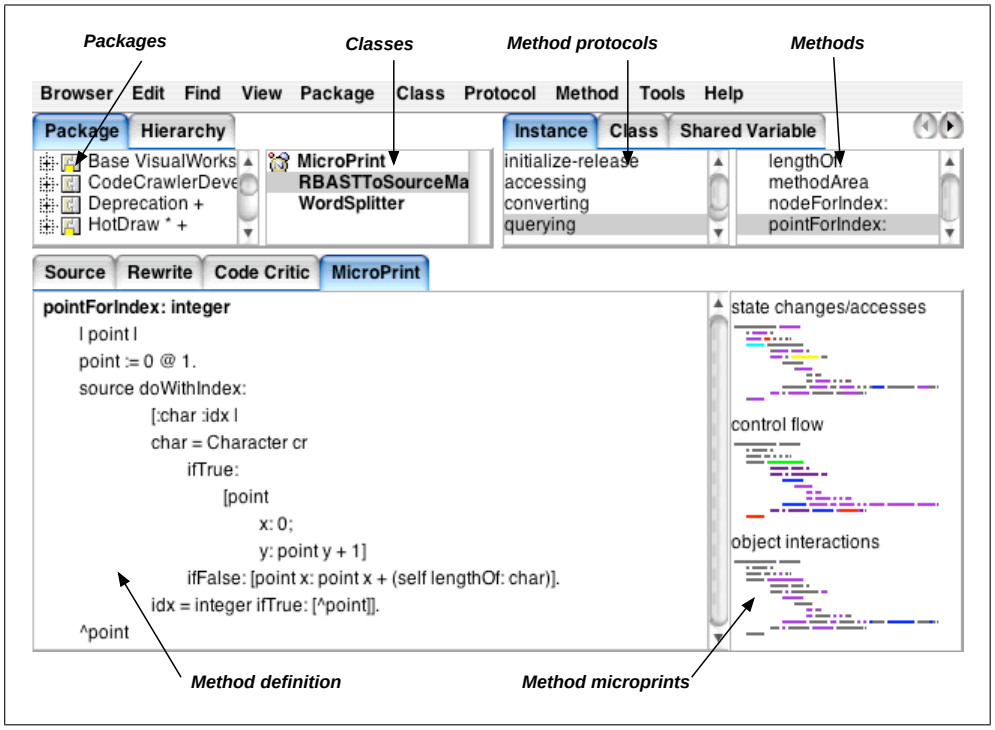

Fig. 10. Microprints integration in a development environment.

5.1 In a Programming Environment

We extended the VisualWorks Smalltalk class browser to display microprints when it displays methods or groups of methods (called method protocols in Smalltalk). When the browser displays a method, several dedicated microprints are displayed for the method (Figure 10). When the browser displays the various protocols of a class, all the methods in that protocol (such as "accessing", "testing", *etc.*) are displayed using the same but changeable microprint, as shown in Figure 9.

The microprints can be chosen by the programmer according to the information he needs. The programmer can also define other dedicated microprints, by creating a new mapping of Markers (objects used to detect and mark elements of a method) to Colors, such as displaying the "assignment to variable" marker in red, the "conditional marker" in green, ... The programmer can also use the framework to define his own kind of microprints in addition to the existing ones (state access, control flow, object interaction, and the microprints focused on dynamic behavior which are mentioned in section 8). We took care of having an easily extensible framework

for the microprints so someone willing to define new microprints has just to create a new subclass of Marker. It can then be included in all microprints with a color using the same procedure.

5.2 Within Class Blueprints

Class blueprints are semantically enriched call-graphs of all methods in a class [12] whose principle is presented in Figure 11. A class blueprint displays the methods and attributes of classes as nodes of a graph, where the edges are the invocations of methods or the accesses of attributes. Methods are classified in four categories: initialization, public interface, private implementation methods, and accessors (see Figure 11). The nodes of the graph are colored to display semantic information of the represented method. However, even if the programmer gets valuable information, he is often forced to read the code.

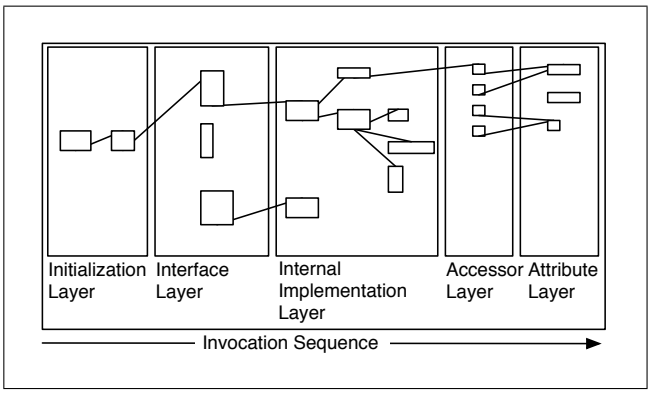

Fig. 11. The class blueprint in a nutshell

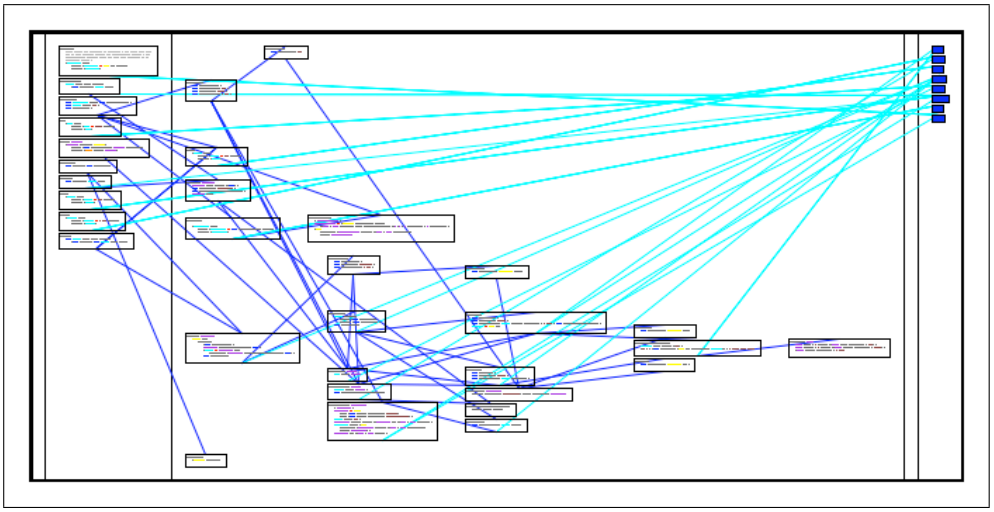

Fig. 12. Class blueprint of the class TestRunner, using the state access microprint

We extended the class blueprint view of CODECRAWLER [12] with microprints. The class blueprint uses colored rectangles to convey semantical information about

the methods and attributes. Microprints extend the class blueprint by displaying a microprint in the rectangle representing a method, allowing the user to have a much better view of what the code does and also to have a *gestalt* impression of a method body without needing to read the source code. The microprint to be displayed is chosen by the user, and can vary between several blueprint views. Figure 12 shows the blueprint of a class, with each method node showing the state access microprint. It is then possible to display the same class blueprint using another microprint, such as control flow, to have another view of the class.

The combination of class blueprints and microprints allows the user to see a lot of methods at the same time. The combined visualization of the call graph allows the reengineer to navigate quite quickly from one microprint to a related one. This vizualisation allows the user to literally "hunt" for particular code patterns (such as the ones enumerated above), to quickly spot areas of classes which needs greater attention. If further insight is needed, the actual code of the methods is just one click away.

For example, we can see from Figure 12 that several methods in the interface layer of class TestRunner are lazy accessors (they present the characteristic cyan-red sequence mentioned above, with an optional yellow word). This insight could be confirmed by switching to a blueprint with the control flow microprint, which would display a red-blue sequence, as shown in Figure 4.

It is also possible to use the class blueprint visualization on a hierarchy of classes, allowing then to see a greater number of methods, as shown in Figure 13. This allows one to grasp collaborations at this higher level, and also to categorize classes based on their behavior. This in turns allows one to easily spot places where the code could be duplicated, thus focusing refactoring efforts.

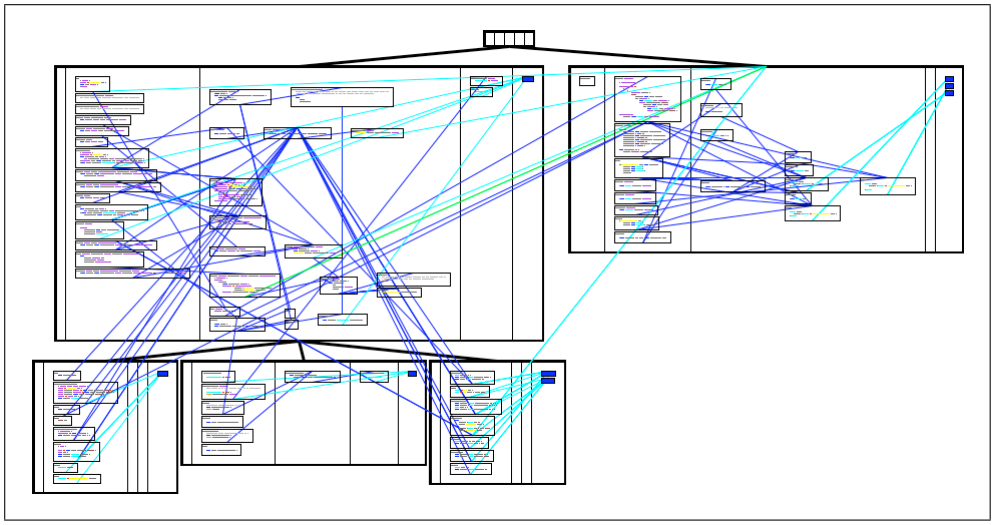

Fig. 13. A hierarchy of classes shown with microprints

6 Discussion

Microprints have the following properties: they take a small amount of space while providing a lot of information, they are non-intrusive and do not modify the source code. They support the identification of visual patterns such as red fragments indicating returns or exception handling, or green fragments indicating loops. They also preserve code indentation, keeping code familiarity and allowing the programmer to map the microprint to the method with better ease.

When looking at a single method, the advantage of microprints over simple code coloring comes from the fact that code coloring cannot display all the available information due to the limited amount of colors we can use. With microprints several facets of the code can be displayed at once.

One drawback of microprints is that the programmer has to navigate between the code and its microprints. However, microprints being smaller than the methods, scrolling is very rarely needed as said above. Thus the navigation does not involve physical movements. While microprints are really effective when used in combination with class blueprints or for entire class hierarchies (or even lists of methods), it is not sure that they are useful for the understanding of a single method. Smalltalk code is generally less verbose than other languages such as Java or C++ (The average length of methods in Smalltalk is 7 lines [11], one-liners being common). We think that in those languages the microprints will prove even more useful, as their utility scale up with the quantity of code to understand at once. We plan to conduct a real evaluation with other programmers to assess if they find microprints a valuable tool and under which circumstances.

A limitation of the microprints is the way the control flow microprints deals with nested blocks. We stated earlier that we should use a few diverse colors to ease pattern recognition, but nested blocks of codes use shades of purple to distinguish one from another. This solution is not ideal as it introduces interpretation problems at small scales such as the ones used in microprints. It is part of our future work to find a solution to this problem.

The integration of the microprints in our tool CODECRAWLER provides a supplemental level of information that in terms of abstractness resides between the class blueprints and the actual source code. An issue is however the scalability of the visualizations: since they are pixel-based they need a definite amount of screen space, while using a vector-oriented approach one could always scale the visualizations to make them fit in one single screen.

7 Related Work

A similar approach has been implemented in SeeSoft [6], which visualizes a large amount of code using pixel-based representations. SeeSoft provides a much higer level view of the code, (entire programs of up to 50000 lines of code), a role which is taken in our approach by other visualizations. Microprints on the contrary are used in smaller-scale views, and provide much more details from the method level up to the class hierarchy level. Hence microprints can provide several parallel views of the same piece of code, whereas Seesoft tends to provide a single view of all the source code. Moreover, Seesoft being much higer-level, it associates a color to a complete line and does not introduce specific visualization for method semantics or finer-grained entities.

Nassi and Shneiderman proposed flowcharts to represent the code of procedures with greater information density[15]. Warnier/Orr-diagrams describe the organization of data and procedures [8]. Both approaches only deal with procedural code and control-flow. Cross et al.defined and validated the effectiveness of Control Structure Diagrams (CSD) [3] [7], which depict the control-structure and module-level organization of a program. Even if CSD has been adapted from Ada to Java, it still does not take into account the fact that a class exists within a hierarchy and that there is late-binding.

Integrated programming environments provide code coloring functionality. Code coloring is interesting because it directly affects the method text itself and enables to have a single focus point while reading the code. The limits of code coloring is that we cannot have simultaneously different views on the same piece of code. In addition, text coloring does not really scale when several methods have to be understood, since the reader has to scroll or open and switch between different windows. A possible extension of our approach would be to apply one microprint as a code coloring scheme, and display the others on the side as we do now.

Many tools make use of static information to visualize software, such as Rigi [19], Hy+ [2] [14], Dali [10], ShrimpViews [18], TANGO [17], as well as commercial tools like Imagix to name but a few of the more prominent examples. Most publications and tools treat classes or methods as the smallest unit in their visualizations. There are some tools, for instance the FIELD programming environment [16] or Hy+ [2] [14], which have visualized the internals of classes, but usually they limited themselves to showing method names, attributes, *etc*. and used simple graphs without added semantic information.

Arévalo [1] proposes X-Ray views, virtual categorizations of methods according to certain heuristics using concept analysis. Three views are proposed based on the state access, the super and self calls and client accesses. However, there is no visualization per se in X-Ray views. The analysis performed for X-Ray views could

be used to create dedicated microprints.

Class blueprints [12] provides a call-flow based representation of classes. Although class blueprints are enriched with semantical information extracted from method analysis, they do not provide fine-grained method-based information.

CODECRAWLER is also used as a visualization tool for software metrics. Microprints use markers instead of metrics, and work on a smaller scale. A marker can be seen as a binary metric, *i.e.*, a program element can comply to the marker, or it cannot. We use markers instead of metrics due to the constraint that we must use a limited number of colors. Using metrics would involve using shades of color, which will reduce the readability of such small visualizations [3]. Whereas metrics are most of the time assigned to entities such as classes, methods or packages, microprints are marking program elements inside a method parse tree, such as references to variables or method calls.

Dekel uses Concept Analysis to visualize the structure of the class in Java and to select an effective order for reading the methods and reveal the state usage [4]. However little information is extracted and the developer has to understand how to read concept lattices in connection with source code.

8 Conclusion and Future Work

In this paper we present microprints, pixel-based representations of the methods and their bodies. We presented three dedicated microprints that each target a different understanding goal. We have also shown how the microprints have been integrated in a commercially available development environment (Cincom VisualWorks Smalltalk) and in a known visualization tool, CODECRAWLER. Even if microprints have been developed for Smalltalk code, our belief is that the technique is easily adaptable to other object-oriented languages, given a parser for the target language, and given some dedicated code markers taking into acount their peculiarities.

In the future we would like to display run-time information such as which parts of the methods have been executed and the frequency of this execution. We currently have already an implementation using the following scheme: a dedicated Smalltalk interpreter broadcasts execution events (variable accesses, message sends, exceptions being thrown and caught), and special Markers can mark the code of the method being run. The program code is then exercised by running its test suite with

[3] It is still possible to use metrics with microprints, it is just not recommended. For example, one of the microprints visualizing dynamic behavior mentioned in section 8 uses metrics to visualize how often a piece of code is executed

this interpreter. The implementation is hovewer not mature enough, and consistent coloring have yet to be found. In addition, this kind of microprint is less portable than the ones described here. Another use of dynamic information we envision is to display when exceptions are raised and caught at run-time by the interpreted code.

Moreover, we want to validate the usefulness of the microprints in an industrial context by releasing the software to the community of Smalltalk developers and evaluate their feedback to ameliorate the microprints.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project *Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1)*. Thanks to Tudor Girba, Orla Greevy, Cyrus Hall, and Mircea Lungu for their comments.

References

[1] Gabriela Arévalo. Understanding behavioral dependencies in class hierarchies using concept analysis. In *Proceedings of LMO '03 (Langages et Modeles à Objets)*, pages 47–59. Hermes, Paris, January 2003.

[2] Mariano P. Consens and Alberto O. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2*, pages 511–516, 1993.

[3] James H. Cross II, Saeed Maghsoodloo, and Dean Hendrix. Control Structure Diagrams: Overview and Evaluation. *Journal of Empirical Software Engineering*, 3(2):131–158, 1998.

[4] Uri Dekel. Revealing JAVA Class Structures using Concept Lattices. Diploma thesis, Technion-Israel Institute of Technology, February 2003.

[5] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-Oriented Inspection in the Face of Delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[6] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[7] Dean Hendrix, James H. Cross II, and Saeed Maghsoodloo. The Effectiveness of Control Structure Diagrams in Source Code Comprehension Activities. *IEEE Transactions on Software Engineering*, 28(5):463–477, May 2002.

[8] David A. Higgins and Nicholas Zvegintzov. *Data Structured Software Maintenance: The Warnier/Orr Approach.* Dorset House, January 1987.

[9] Dean F. Jerding and John T. Stasko. The information mural: Increasing information bandwidth in visualizations. Technical Report GIT-GVU-96-25, Georgia Institute of Technology, October 1996.

[10] Rick Kazman and S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, April 1999.

[11] Edward J. Klimas, Suzanne Skublics, and David A. Thomas. *Smalltalk with Style*. Prentice-Hall, 1996.

[12] Michele Lanza and Stéphane Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of OOPSLA '01 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 300–311. ACM Press, 2001.

[13] Michele Marchesi and Giancarlo Succi, editors. *Extreme Programming and Agile Processes in Software Engineering*. Springer, 2003.

[14] Alberto Mendelzon and Johannes Sametinger. Reverse engineering by visualizing and querying. *Software — Concepts and Tools*, 16:170–182, 1995.

[15] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8), August 1973.

[16] Steven P. Reiss. Interacting with the field environment. *Software — Practice and Experience*, 20:89–115, 1990.

[17] John T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.

[18] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and Documenting Software Structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275–284. IEEE Computer Society Press, 1995.

[19] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse enginnering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[20] Edward R. Tufte. *Envisioning Information*. Graphics Press, 1990.

[21] Colin Ware. *Information Visualization*. Morgan Kaufmann, 2000.

[22] Norman Wilde and Ross Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.