Scoping Changes with Method Namespaces

Alexandre Bergel

ADAM Project, INRIA Futurs
Lille, France
alexandre.bergel@inria.fr

Abstract. Size and complexity of software has reached a point where modular constructs provided by traditional object-oriented programming languages are not expressive enough. A typical situation is how to modify a legacy code without breaking its existing clients.
We propose *method namespaces* as a visibility mechanism for behavioral refinements of classes (method addition and redefinition). New methods may be added and existing methods may be redefined in a method namespace. This results in a new version of a class accessible only within the defining method namespace. This mechanism, complementary to inheritance in object-orientation and traditional packages, allows unanticipated changes while minimizing the impact on former code.
Method Namespaces have been implemented in the Squeak Smalltalk system and has been successfully used to provide a translated version of a library without adversely impacting its original clients. We also provide benchmarks that demonstrate its application in a practical setting.

1 Introduction

Managing evolution and changes is a critical part of the life cycle of all software systems [BMZ+05, NDGL06]. In software, changes are modeled as a set of incremental code refinements such as class redefinition, method addition, and method redefinition. Class-based object-oriented programming languages (OOP) models code refinements with subclasses that contain behavioral differences. It appears that subclassing is well adapted when evolution is anticipated. For example, most design patterns and frameworks rely on class inheritance to express future anticipated adaptation and evolution. However, subclassing does not as easily help in expressing unanticipated software evolution [FF98a, BDN05b].

The work presented in this paper is a revival of selector namespace implemented in Smallscript[1] by Dave Simmons.

In recent years, researchers have produced numerous constructs and languages extensions to better modularize software changes. Class extensions *à la* CLOS, Smalltalk enable and ObjectiveC method addition and redefinitions by a package different from the package that defined the method's class. This mechanism has been recently adopted by AspectJ. However, the visibility of those extensions is global which lead to conflicting situations between concurrently developed extensions. This visibility problem is the focus of this paper.

[1] www.smallscript.net

This paper presents *Method Namespaces*, a modular construct for OOP languages that consists a single parent namespace and a set of method definitions. When a method is defined, an associated annotation identifies its namespace. Method definitions in a namespace are accessible only within the defining namespace and its children. Since a method namespace has one parent namespace, a set of namespaces is structured as a tree. Method namespaces enable several versions of the same method to coexist with minimal confusion as to which version will be invoked.

The runtime semantics of method namespaces is driven by its *pellucid property*: the system behaves as if methods defined in an ancestor namespaces were directly defined in the child namespace. Thus, a method namespace represents the scope for a particular version of a group of methods.

By encapsulating a set of method implementations, a namespace has the ability to define a software refinement that may crosscut several classes. Such a refinement is visible only within the method namespace in which it is defined and in its descendent. Thanks to the scoping mechanism, conflicts between namespaces are eliminated.

The paper is structured as follows: Section 2 illustrates the issues encountered by the community around the Squeak Smalltalk when translating its libraries. Section 3 describes the method namespace model and its properties. It also shows how the issues related to the translation are solved. Section 4 presents the implementation of Method Namespaces in the Squeak Smalltalk system. Section 5 presents the related work. Finally, Section 6 concludes this paper and outlines our future work.

2 The Need for Scoping Changes

Multilingual support. Managing unanticipated software evolution has been a software engineering problem that attracts a large interest [MFH01, FF98b, TB99, BMZ^{+}05]. This section describes a situation that the Squeak[2] [IKM^{+}97] community has encountered when adding a multilingual support.

Before this translation effort, English was ubiquitous in all object descriptions and tool implementations. For example, the class Object defines a method printOn: aStream as follows:

```
Object>>printOn: aStream
  "Append to the argument, aStream, a sequence of characters that
  identifies the receiver."

  | title |
  title := self class name.
  aStream
    nextPutAll: (title first isVowel ifTrue: ['an '] ifFalse: ['a ']);
    nextPutAll: title
```

This method simply prepends an article before the name of the object's class. Printing an object into a stream invokes this method. For example, the printing of an instance of a Car class returns a Car and an instance of Object returns an Object.

[2] www.squeak.org

Multilingual support is achieved by invoking the message translate on each used string. For example, a multilingual version of printOn: aStream contains 'an ' translated and 'a ' translated instead. This translate method is defined on the String class and performs a lookup into a global translation dictionary; 'a ' and 'an ' will translate into 'un ' for a French translation.

Need for coexisting versions. Parts of Squeak were refactored by adding such translate message send to strings. Whereas this mechanism may be sufficient for menus and windows title, it fails whenever a particular convention or sentence structure has to be taken into account. For example, the method Date>>fromString: aString is used to convert a textual date representation into an instance of the Date class. A typical illustration is Date fromString: 'August 12, 2007'. The parameter of fromString: aString may match different patterns such as '8/12/2007'. The parameter aString has to follow the American english way of writting dates (month followed by the day, then the year). In most European countries, a date starts with the day followed by the month and the year as in 12/8/2007. Implementing a multilingual mechanism cannot be achieved by translating strings only. A different parsing for dates has to be employed instead. A similar situation occurs with reading and printing of time. Hours range from 0 and 23 or from 1 and 12.

The two fromString: aString methods defined on Date and Time have to be rewritten in order to have a proper French translation. However, the Squeak runtime makes a heavy use of those methods. Parts related to the system event logging and source code change versioning rely on the original version of the Date and Time classes. For example, the DPRecentlyModified class augments the system code browser by informing the programmer about the recently modified methods. Determining whether a method is recent is achieved by comparing method time stamps. Those time stamps are extracted as a string from the list of changes separately stored in a file. Next, instances of Date and Time are obtained using the fromString: aString conversion methods defined in those classes. As a consequence, replacing the American English version of fromString: aString with the French version will not correctly handle legacy time stamps that are stored (using the American English format) since a method modified on January 12 has been stored under the '1/12/2007' in the list of changes. When read back by the DPRecentlyModified class, this time stamp appears to be December 1st using the French format. The list of recently modified methods will therefore be incorrect.

This example demonstrates the difficulty of updating parts of a system while preserving original system behavior. A complete multilingual version of Squeak must handle different time and date formats properly. However the Squeak core implicitly assumes that the format of already stored time stamps follows the American English conversion methods, which is not the case if modifications to Date and Time are universally visible.

Limitation of class inheritance. Creating two new classes FrenchDate and FrenchTime that would subclass Date and Time, respectively, is not satisfactory since all references to those classes will have to be reviewed and possibly updated to reference the new subclasses. For example, among the 5002 classes present in the development version of

the Squeak system[3], Date is referenced 80 times and Time 236 times. Subclassing these two classes will imply heavy rewritings to make the French subclasses used by some, *but not all* of the legacy code. This limitation of inheritance to express incremental refinements has been extensively described in the literature [FF98a, BDN05b].

Problem analysis. The solution to address the issue described above will have to exhibit four properties:

– *Coexistance of multiple implementations.* Adding a multilingual support as described above is a typical situation where at least two implementations for a same method must coexist: one needed by the Squeak runtime to deal with system notification and logging and another to be used by the client program.
– *Scoping of implementations.* Incorporating the French translation in the Squeak system is not practical because of the global impact this translation has. The French version of methods should be scoped in order to avoid any unanticipated impact on the base system.
The application user that requires the French translation should run in a scope where the changes related to the translation are effective. Outside this scope, those changes should not be in effect: the Squeak versioning and changes logging mechanisms should stick to the American English way of storing dates and times to avoid confusion with the legacy data.
– *Class identity and class refinement.* As we have seen, subclassing is not satisfactory because a newly created class cannot be used by legacy code. Refining a class by adding new versions for its methods should preserve the identity of the class.
There are almost 8000 instances of the Date and Time classes living in the Squeak runtime image. A large part of those instances define the time stamps of existing methods. Ideally, these instances would be displayed accordingly to the wanted language (*i.e.,* American English or French). Living instances should benefit from refinement defined on their classes.
– *Method implementation and scope.* A scope should define one unique implementation for a given method. Having more than one method implementations accessible in the same scope requires one to decide at runtime which implementation to choose upon message sending, a problem that method namespaces nicely avoids.
Our previous experience [BDN05b] shows that having more than one accessible version of a method in a given scope brings a significant amount of complexity resulting in a system that may be difficult to understand and runtime slowdown. Allowing only one version of a method in a given scope enables the graph of scope to be flattened, thus making it easier to understand.

3 Method Namespaces

This section presents *Method Namespaces*, a language construct to scope behavioral class refinements such as addition and redefinition of methods. A refinement consists of a group of methods, in which each method may either replace a previous method

[3] Squeak 3.10 - 7130dev07.07.1

implementation, or add to the interface of the refined class. Method Namespaces are a mechanism orthogonal and complementary to class inheritance and package mechanisms.

3.1 Method Namespaces as Container of Refinements

A *method namespace* is a container for class refinements (*i.e.,* method addition and redefinition). It encapsulates a set of incremental class refinements and limits the impact of those refinements to a well delimited part of the software program considered.

Method namespaces are an extension of the object-orientation paradigm intended to provide a *refinement* capability, which complements the *specialization* offered by inheritance along classes and *packaging* offered by package systems.

A method namespace encapsulates class refinements by defining a set of method implementations. Such a namespace has a single parent namespace (except for Default which has no parent). Default is always present. Each method implementation belongs to a namespace.

A method namespace defines a scope in which code may be executed. The behavior of this code is defined by the method implementations provided in this scope and its ancestors. A namespace may hide implementations provided by its ancestors by simply redefining them. A namespace has one parent at the most.

As described in the subsequent sections, the rational to have only one parent is to give namespaces the ability to be compiled away without requiring complex composition operators.

Example. Figure 1 makes use of a method namespace to contain the refinements related to the French translation of the system. In the default method namespace, dates representation follows the *MM-DD-YYYY* format, hours range from 1 to 12, and an instance of Object is printed as 'an Object'. In the French namespace dates follow a *DD-MM-YYYY* format, hours range from 0 to 23, and an instance of Object is printed as 'un Object'. Figure 1 illustrates how a method namespace allows multiple versions of a system to coexist. The result of an expression such as Object new printString depends on the namespaces in which it is evaluated.

Definition of method namespaces. A namespace is defined by providing a new name and the name of the parent namespace:

```
MethodNamespace
  create: #French
  parent: #Default
```

Classes do not belong to a namespace, however their methods do. For example, the class Date is defined as[4]:

[4] To keep the description short, definition Date is shortened by removing class variables and pool declarations.

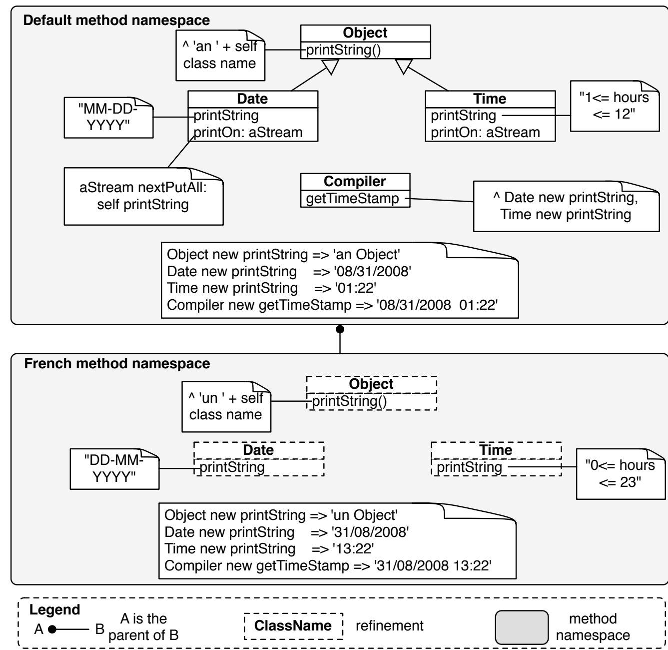

Fig. 1. A set of refinements with Method Namespaces.

```
Object subclass: #Date
  instanceVariableNames: 'start duration'
  category: 'Kernel-Chronology'

Date compile: 'printString
  "Return a string that follows the MM-DD-YYYY pattern"
  ^... '
  namespace: #Default
```

Date is defined as a subclass of Object and has two instance variables. It belongs to the category Kernel-Chronology[5].

The method printString is defined in the class Date. This method belongs to the Default namespace. In the French namespace, this method is redefined:

```
Date compile: 'printString
  "Return a string that follows the DD-MM-YYYY pattern"
  ^... '
  namespace: #French
```

[5] Class category is a rudimentary classification mechanism for classes provided by Smalltalk.

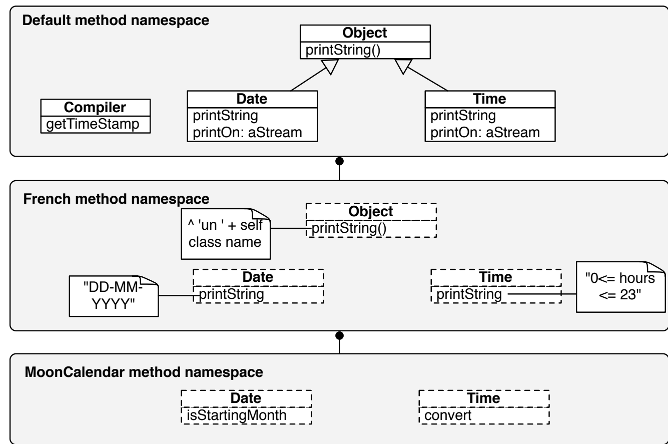

Fig. 2. The MoonCalendar namespace refines Date and Time futher.

Locality of refinements. Refinements are local to the method namespace in which they are defined. We refer to this property as the *locality* of refinements. Outside a method namespace, refinements are not accessible.

3.2 Method namespace at runtime

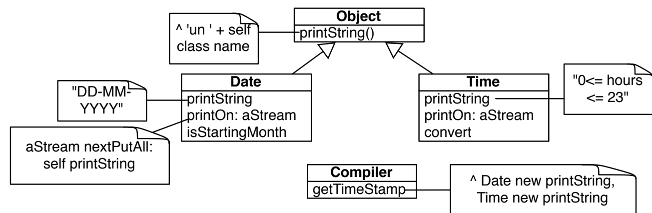

Fig. 3. Pellucid view from the MoonCalendar namespace.

Defining a namespace child allows for further refinements on its parent by adding or redefining methods. Figure 2 proposes a variant of the example in which the French namespace is further refined by MoonCalendar which augments Date with a isStarting-Month method and Time with a convert method.

Control flow and namespace. A thread may be created in a method namespace. As a consequence, threads define the granularity of the scope of a namespace. One or more

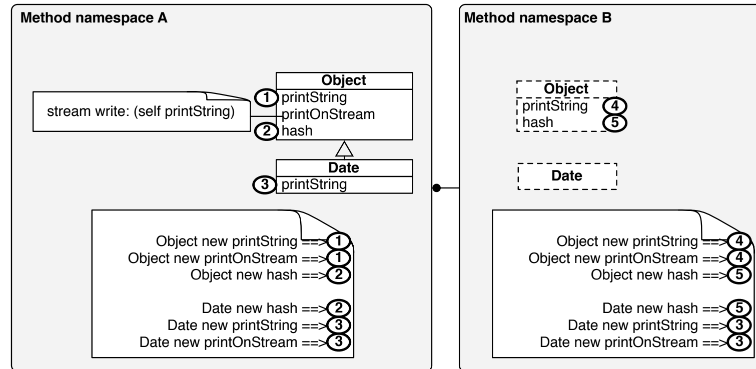

Fig. 4. The method lookup algorithm goes through the import link before inheritance.

threads may live in the same namespace. This enables a set of threads to share the same version of a system.

Pellucid property. Conceptually, methods defined along the chain of parent namespaces may be inlined in the namespace that will be used at runtime. Within one thread, it is as if namespaces are compiled away. Import links between namespaces defines an ordering: refinements of a namespace override the definitions in parent namespaces. Figure 3 shows the result of a flattening of the MoonCalendar namespace. This figure is the namespace compiled-away version of Figure 2.

Import has precedence over inheritance. A method lookup is triggered at each message sent. If the current namespace does not provide an implementation for the selector, then the lookup will be repeated (recursively) in the parent namespace.

Figure 4 illustrates this situation. On the left hand side, invocations of printString and printOnStream use the implementation provided by namespace 1. On the right hand side, invocations of printOnStream on an instance of Object leads to an execution of the new version of this method (provided by namespace 2). Invoking printString on a date uses the implementation of namespace 1 since printString is overridden in Date, its superclass.

Object creation. An object is intrinsically associated to the class it was created from. However, the object is not associated to a particular version of this class, even in presence of multi-threading. Objects may be shared between different threads that live in different namespaces. In that case, the same object may behave differently according to which thread it is used in.

4 Implementation

Method namespaces are implemented in Squeak [IKM+97], an open-source Smalltalk dialect. This section describes the key implementation aspects of this work. Whereas most of the presented code follows the syntax of Squeak, we hope this will not hamper the reader from understanding the general approach.

4.1 The Squeak execution model

Smalltalk promotes a ubiquitous reification of most important aspects of the runtime [Riv96]. For example, classes, dictionaries of methods, and method definitions are each first-class objects making them subject to standard object compositions and manipulations rules.

A compiled method is an ordered collection of bytecode instructions. It is the elementary support for runtime execution in Smalltalk. The behavior of a class is defined by a set of associations *selector* → *compiled method*. In the Smalltalk terminology, a selector is a symbol (à la Lisp) is a method name. When a message is sent to an object, the Squeak virtual machine (VM) looks up selector in the receiver's class's method dictionary (and so on to superclasses). If the selector is found, the associated compiled method is executed by the VM.

4.2 Dynamic dispatch

One feature of the Squeak virtual machine is that it reifies a message whenever a plain object has replaced a compiled method in a method dictionary. When a method has been fetched, the virtual machine checks whether the retrieved association's value is a compiled method. If it is not a compiled method, then the message run:with:in: is sent to the retrieved object [BD06]. Our implementation of method namespaces is based on this message reification. The idea consists of using a *dispatcher* that looks up a method version according to the encapsulating method namespace.

In Figure 5, the class Object is refined with a second version of printString. The left hand side of the figure describes this situation from a conceptual point of view, whereas the right hand side presents the memory organization. The Object class is an object that references a method dictionary, which represents the behavior of this class. Object is extended with two methods, printString is redefined and description is a new method. The method dictionary has 3 entries therefore:

- printOnStream is associated with a compiled method since this method is defined in the default namespace and has not been redefined. No dispatcher needs to be used since the default namespace is the top most parent of all namespaces, therefore always accessible.
- Since two versions of printString are coexisting, a dispatcher is employed to select the right version of printString upon invocation.
- One single version of description is present, however a dispatcher has to be used to restrict invocations of description to the French namespace.

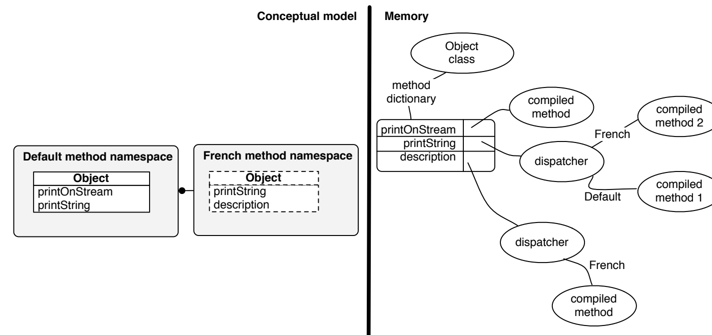

Fig. 5. Coexistance of method versions uses a dispatcher.

4.3 Method version lookup

The version selection process of a particular method is performed after a message invocation has been looked up. This is achieved by the dispatcher. The method run:with:in: is in charge of the method version selection:

```
Dispatcher>>run: sel with: arguments in: receiver
  | currentNamespace cm |
  currentNamespace := Processor activeProcess namespace.
  cm := currentNamespace
            retrieveCompiledMethodFromSelector: sel inClass: theClass.
  cm ifNil: [ˆ receiver
                perform: sel
                withArguments: arguments
                inSuperclass: theClass superclass].
  ˆ cm valueWithReceiver: receiver arguments: arguments
```

As mentioned in Section 3.2, the control flow of an application is associated with a namespace. A thread, called process in the Smalltalk terminology, knows which namespace it was triggered in. The expression Processor activeProcess namespace returns the namespace of the current process. Processor is a reification of the process scheduler and activeProcess returns the one currently active.

The method retrieveCompiledMethodFromSelector:inClass: walks over the import chain of namespaces until a method version is found. It returns nil if none has been found. If no implementor for the selector is found, the method is resent, starting from the superclass. Note that ˆ is the return statement. If an implementation is found, the compiled method is evaluated with the appropriate arguments and receiver. (Note that 'self' in this method refers to the dispatcher, and not to the receiver of the message and self-references contained in the compiled method refer to the object receiver, the object on which the message has been invoked.)

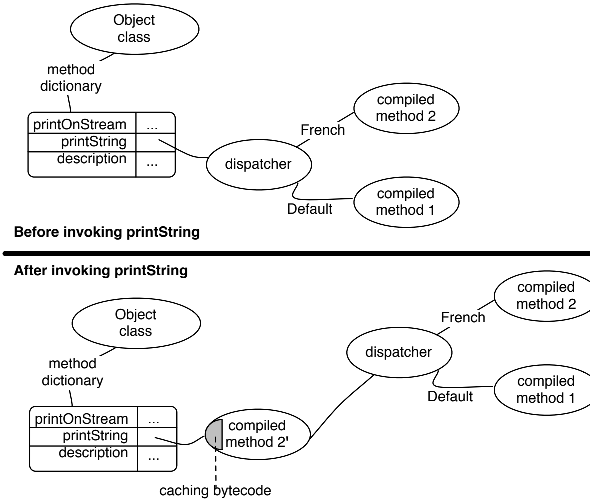

Fig. 6. Caching effect when invoking printString from the French namespace.

4.4 Caching strategy

Message reification as described above is relatively costly. Reifying a message is roughly 5 times slower than invoking a message directly. We have developed a namespace caching strategy that consists of prepending the necessary set of bytecode instructions at the beginning of a scoped method to check whether this method is invoked from the thread it has been previously cached from. The assumption favored by this caching strategy is *a method will most of the time be invoked from the same method namespace*.

After a message reification, a copy of the compiled method is prepended with the caching bytecode. Then, this new compiled method replaces the dispatcher in the method dictionary. Figure 6 illustrates this situation. The Default and French namespaces each have their own version of printString, *compiled method 1* and *compiled method 2*, respectively. Assuming an invocation in French, a copy of *compile method 2* is created, then the caching bytecode is added at the beginning of it. This new method is inserted in the method dictionary of Object.

The cache become invalidated when printString is invoked from a method compiled inDefault namespace and *compiled method 2* is removed from the method dictionary.

The caching bytecode corresponds to the compiled version of the following code:

```
(Processor activeProcess namespace == <namespace>) not
  ifTrue: [
    <myClass> methodDictionary at: <methodName> put: <dispatcher>.
    ^<dispatcher> run: <methodName> with: <args> in: self]
```

<namespace>, *<myClass>*, *<methodName>*, *<dispatcher>*, *<methodName>* are replaced when the cache is created by the current method namespace, the class that contains the method, the name of the method, and the dispatcher, respectively.

The length of this cache is about 32 bytecode instructions, however only 6 bytecode instructions are executed when the cache is valid. In our implementation, this cache is generated with ByteSurgeon [DDT06], a framework to operate on the bytecode in Squeak.

4.5 Benchmarks

We have measured the time overhead over several benchmarks.

Micro-benchmark. Since the length of the cache code is constant in size (32 bytecode instructions), prepending it on a very short method results in an expensive invocation. The extreme case is when a method returns the self value. A method, like yourself, that simply returns an immediate value is optimized in the Squeak virtual machine by tagging the method:

```
Object>>yourself
  ^self
```

In the original Squeak, this method does not contain any bytecode. In the version of Squeak supporting method namespace this method is 32 bytecode instructions long.

On a MacBook[6], the virtual machine (version 3.8.12beta4U) executes the yourself method 19,379,844 times per second. The number of executions per second is 4,061,738 in the Method Namespaces aware version of Squeak. A factor of 4.77 reflects the overhead of Method Namespaces for this particular method.

As a second micro-benchmark, let us consider a method that is not a tagged method. The method callYourself calls yourself. The complexity of the computation of the method is relatively low since it simply sends yourself to self:

```
Object>>callYourself
  ^self yourself
```

The number of executions per seconds of callYourself is 7,639,419 in the original Squeak. In the Method Namespaces aware version of Squeak the number of calls per second rises to 2,068,680. A factor of 3.69 reflects the overhead of Method Namespaces for this particular method.

As a third and the last micro-benchmark, we evaluate the cost of Method Namespaces on an “average” method. The average number of lines of code is Smalltalk is about 7 lines of code. The method dismissButton on the class AllScriptsTool is a typical example. This method is executed 2793 times per second. In the Method Namespaces aware version of Squeak it is executed 2729 times per second in the average. For this particular method the overhead is about 2%. Although we cannot draw any conclusion from this third example, this examples shows that the cost of the version selection performed by the dispatcher is at negligible cost of the time taken to perform the computation of the method.

---

[6] 1.83 GHz Intel Core 2 Duo with 1 GB 667 MHz DDR2 SDRAM

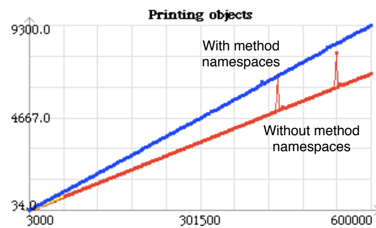

Fig. 7. Printing objects translated in French (with method namespace) versus in American English (without method namespaces) .

Macro-benchmark. We have translated Squeak as described in previous sections. The overhead when displaying the French translation vs. the American English version is depicted by Figure 7. The X-axis represents the number of times printString message is performed, and the Y-axis represents the time necessary to performs those printings in milliseconds.

The benchmark consists of displaying a large set of objects on the screen. The upper curve represents the time spent in displaying them in French, and the lower curve in American English using the original non modified Squeak (where the two peaks probably stem from the garbage collection activity). Figure 7 shows the overhead of method namespace in an activity that involves a large part of the streaming and collection library of squeak. The ration between using and not using method namespaces is 1.34.

5 Related Work

Aspect-oriented programming. AOP is a programming technique where concerns that cut across a software system can be described in clear statements so that the underlying design intent remains clear in the source code. Separation of cross-cutting concern is achieved through the use of a set of pointcut and advice descriptions.

With its notion of inter-type, AspectJ allows class members to be separated from the class definition by being defined in an aspect. Whereas with method namespaces a class can be refined in two namespaces with two methods having the same name, with Aspect/J conflicts are not allowed: two aspects cannot define two methods having the same name on the same class. This kind of extension does not allow redefinition and consequently does not help in supporting unanticipated evolution.

Virtual classes. Virtual classes were originally developed for the language BETA [KMMPN87], primarily as a mechanism for generic programming rather than for extensibility [MMP89]. Keris [Zen02], Caesar [AGMO06], and gbeta [Ern99] offer such a mechanism, where method and class lookup are unified under a common lookup algorithm. In a similar way that a method is looked up according to an instance, a class is looked up according to an instance (*i.e.,* an encapsulating class). With such a unification of method and class lookup, the role of a class is overloaded with semantics of packages

and objects constructor. With namespace, we keep the original meanings of class and package separate.

Open classes. MultiJava [MRC03] is an extension of Java that supports open classes and multiple method dispatch. An open class is a class to which new methods can be added. Method redefinitions are not, however, allowed: an open class cannot have one of its existing methods refined.

ECMAScript and Smallscript. A notion of namespace similar to Method Namespace has been recently added to ECMAScript[7] and Smallscript[8]. This mechanism is intended to ease the evolution of intensively used libraries. A new version of a method may be defined in a namespace. A version is *statically* associated to a method invocation. The version is the one provided by the namespace where the call occurs. As a consequence, new method versions cannot be used from legacy code. We qualify this behavior as *non-reantrant* [BDN05a].

This is a major difference with Method Namespace and Classboxes: former code may benefit from new version of methods.

Expanders. Expanders [WSM06] is an extension of the OO paradigm that support object adaptation. It allows classes to be non-invasively updated with new methods, fields and interfaces. By importing a set of expanders, a client may 'adapt' some classes to its particular need.

Expanders rely exclusively on static type annotation. It has to be statically determined whether a feature invocation makes use of an expander or not. Expanders follow a different philosophy from Method Namespaces since we mainly focus on dynamically typed languages.

Classboxes. A classbox [BDN05b] is a module containing scoped definitions and import statements. Classboxes define classes, methods and variables. Imported declarations may be extended, possibly redefining imported methods.

Classboxes gives a new semantic to the method lookup algorithm to achieve the method version selection for a given set of interacting classboxes. Our experience with Classboxes suggests that such a lookup tends to be complex since several versions of a same class may have to be accessible within the same classbox.

By proposing a pellucid property, complexity of Classboxes has been removed in Method Namespace.

Context-oriented programming. ContextL [CH05] is an extension to the Common Lisp Object System provides means to associate partial class and method definitions with layers and to activate and deactivate such layers in the control flow of a running program. When a layer is activated, the partial definitions become part of the program until the later is deactivated.

[7] http://wiki.ecmascript.org
[8] http://www.smallscript.net

ContextL's Layers cannot be flattened, the pellucid property is therefore not supported. The effect is that no assumption can be statically made upon the dynamic selection of a method version. As an example, let us assume a set of layers where each provides an implementation of a method display-object. When a programmer writes (display-object myObject), which sends the message display-object to myObject, it may not be possible to statically determine which version of display-object will be invoked since no assumption can be made upon the current control flow.

This is pretty much the same situation with polymorphism and with Classboxes. Our experience with Classboxes showed that this dynamic selection of method (added to polymorphism) lowers the comprehensibility of the overall system.

View-oriented programming. CorbaViews [MMS02] promotes the view-oriented paradigm that considers application objects as a core functionality to which state and behavior (*views*) are added and retracted on demand during run-time. A view may be added and removed during run-time, making objects support different interfaces while an application is executing.

View-oriented programming does not employ a scoping mechanism to limit a view and operates at the level of the object, instead of the class. Views are individually attached to objects. Methods Namespaces enables classes to be refined under a specified scope without affecting the instantiation mechanism. As a result, Views would give a more complex and more invasive solution to our translation example.

6 Conclusion

Method Namespaces are a programming language construction aimed to scope behavioral changes such as method additions and redefinitions. Such a namespace is a container for method definitions and versions. A method may be invoked only within the namespace in which it is defined and in children namespaces. Outside those namespace, the method cannot be invoked.

Our work is motived by having a non-invasive translated version of a set of classes. We have presented Method Namespaces as an elegant solution to solve this problem in a non-invasive way (without implying costly refactoring). We also have demonstrated that Method Namespace can be put in practice with an overhead factor of 1.34, suitable to a large range of application domains.

As a future work, we plan to widen the range of domain application of Method Namespace to security and interoperability between Smalltalk dialects.

Acknowledgements. We would like to thank Marcus Denker for his help in implementing the method caching with ByteSurgeon. We also thank James Foster for the extensive discussion we had and his comments on this paper. We acknowledge Mathieu Suen for his review on an earlier draft, Yann-Gaël Guéhéneuc, Robert Hirschfeld and Houari Sahraoui for their comments on the idea described in this paper.

References

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135 – 173, 2006.

[BD06] Alexandre Bergel and Marcus Denker. Prototyping languages, related constructs and tools with Squeak. In *Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.

[BDN05a] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 11(10):1613–1644, November 2005.

[BDN05b] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of OOPSLA'05*, pages 177–189, 2005. ACM Press.

[BMZ+05] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice*, pages 309–332, 2005.

[CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05*, October 2005. ACM Press.

[DDT06] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

[Ern99] Erik Ernst. *gbeta — a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[FF98a] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the third international conference on Functional programming*, pages 94–104. ACM Press, 1998.

[FF98b] Matthew Flatt and Matthias Felleisen. Units: Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings of OOPSLA '97*, pages 318–326. ACM Press, November 1997.

[KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The BETA programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, Cambridge, Mass., 1987.

[MFH01] Sean McDirmid, Matthew Flatt, and Wilson Hsieh. Jiazzi: New age components for old fashioned Java. In *Proceedings of OOPSLA 2001*, pages 211–222, October 2001.

[MMP89] Ole Lehrmann Madsen and Birger Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89*, volume 24, pages 397–406, October 1989.

[MMS02] Hafedh Mili, Hamid Mcheick, and Salah Sadou. Corbaviews – distributing objects that support several functional aspects. *Journal of Object Technology*, 1(3):207–229, August 2002.

[MRC03] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed multijava: balancing extensibility and modular typechecking. In *Proceedings of OOPSLA'03*, pages 224–240. ACM Press, 2003.

[NDGL06] Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, and Adrian Lienhard. Analyzing, capturing and taming software change. In *Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.

[Riv96] Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996.

[TB99] Lance Tokuda and Don Batory. Automating three modes of evolution for object-oriented software architecture. In *Proceedings of COOTS '99*, May 1999.

[WSM06] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proceedings of OOPSLA'06*, pages 37–56. ACM Press.

[Zen02] Matthias Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.