IWST 2012

Proceedings of the 4th edition of the International Workshop on Smalltalk Technologies

In conjunction with the
20th International Smalltalk Joint Conference

Gent, august 2012

Goals and scopes

•The goals of the workshop is to create a forum around advances or experience in Smalltalk and to trigger discussions and exchanges of ideas. Participants are invited to submit research articles. We will not enforce any length restriction. However we expect papers of two kinds:

–Short position papers describing emerging ideas.

–Long research papers with deeper description of experiments and of research results.

•We welcome contributions on all aspects, theoretical as well as practical, of Smalltalk related topics such as:

–Aspect-oriented programming,

–Design patterns,

–Experience reports,

–Frameworks,

–Implementation,

–new dialects or languages implemented in Smalltalk,

–Interaction with other languages,

–Meta-programming and Meta-modeling,

–Tools

Publication

•Both submissions and final papers must be prepared using the ACM SIGPLAN 10 point format. Templates for Word and LaTeX are available at http://www.acm.org/sigs/sigplan/authorInformation.htm

•Authors of the best accepted papers will also be invited to submit extended versions for publication in a special issue of Elsevier "Science of Computer Programming: Methods of Software Design: Techniques and Applications"

•Please submit your paper through EasyChair on http://www.easychair.org/conferences/?conf=iwst2012

Program Chairs

Program Committee

Table of Contents

Author Index

Keyword Index

On the Integration of Smalltalk and Java

Practical Experience with STX:LIBJAVA

Marcel Hlopko
Czech Technical University in Prague
marcel.hlopko@fit.cvut.cz

Jan Kurš
Software Composition Group, University of Bern
kurs@iam.unibe.ch

Jan Vraný
Czech Technical University in Prague, eXept Software AG
jan.vrany@fit.cvut.cz

Claus Gittinger
eXept Software AG
cg@exept.de

Abstract

After decades of development in programming languages and programming environments, Smalltalk is still one of few environments that provide advanced features and is still widely used in the industry. However, as Java became prevalent, the ability to call Java code from Smalltalk and vice versa becomes important. Traditional approaches to integrate the Java and Smalltalk languages are through low-level communication between separate Java and Smalltalk virtual machines. We are not aware of any attempt to execute and integrate the Java language directly in the Smalltalk environment. A direct integration allows for very tight and almost seamless integration of the languages and their objects within a single environment. Yet integration and language interoperability impose challenging issues related to method naming conventions, method overloading, exception handling and thread-locking mechanisms.

In this paper we describe ways to overcome these challenges and to integrate Java into the Smalltalk environment. Using techniques described in this paper, the programmer can call Java code from Smalltalk using standard Smalltalk idioms while the semantics of each language remains preserved. We present STX:LIBJAVA — an implementation of Java virtual machine within Smalltalk/X — as a validation of our approach.

Categories and Subject Descriptors H.3.3 [*Programming Languages*]: Language Constructs and Features—Language Interoperability; H.3.4 [*Programming Languages*]: Processors—Interpreters, Virtual Machines

General Terms Language Interoperability

Keywords Language Interoperability, Smalltalk, Java

1. Introduction

Without doubt, the Java programming language has become one of the most widely used programming languages today. A significant amount of code is written in Java, ranging from small libraries to large-scale application servers and business applications. Nevertheless, Smalltalk still provides a number of unique features (such as advanced reflection support or expressive exception mechanism) lacking in Java, which makes Smalltalk suitable for many kinds of project. It is a tempting idea to call Java from Smalltalk and vice versa, as it enables the use of many Java libraries within Smalltalk projects.

The idea of Java-Smalltalk integration is not new and has been explored by others in the past. JavaConnect (Brichau and De Roover [1]) and JNIPort (Geidel [4]) use foreign function interfaces to connect to the Java virtual machine and to call a Java code. Bridges, such as VisualAge for Java (Deupree and Weitzel [2]) or Expecco Java Interface Library (Expecco [3]), use proxy objects, which intercept and forward function calls and return result values or handles via an interprocess communication channel. Another more versatile approach, is to execute both languages within the same virtual machine and use a common object representation for both. STX:LIBJAVA, which is presented in this paper, is an example of such an approach: it executes Java within the Smalltalk virtual machine. Another example is Redline

Smalltalk (Ladd [9]), which executes Smalltalk using standard Java virtual machine.

Seamless, easy to use integration of two programming languages consists of various parts. First, it must allow one language to call functions in the other, possibly passing argument objects and getting return values (*runtime-level* integration).

Second, it should support programmer-friendly argument and return value passing between the languages.

Third, it should ideally preserve object identity. The use of replicas or proxy objects can introduce various problems when objects are stored or managed by their identity. This also affects any side effects to such objects when calling functions in the other language.

Finally, it should seamlessly integrate the two languages on the syntactic level, which means that (ideally) no additional glue or marshalling code should be required to call the other language (*language-level* integration).

In the case of Java and Smalltalk, language-level integration raises a number of challenges, due to their different design and semantics. In particular these are:

- Smalltalk uses keyword message selectors, whereas Java uses traditional C-like selectors (virtual function names).
- Java supports method overloading based on static types, whereas there is no static type information in Smalltalk.
- Exception and locking mechanisms differ.

In this paper, we present STX:LIBJAVA, a Java virtual machine (JVM) implementation built into the Smalltalk/X environment. STX:LIBJAVA allows for the program to call Java code from Smalltalk almost as naturally as normal Smalltalk code. We will demonstrate how STX:LIBJAVA integrates Java into Smalltalk and we will describe how it deals with semantic differences.

The contributions of this paper are (i) a new approach of Smalltalk and Java integration, (ii) identification of problems imposed by such an integration and (iii) solutions for these problems and their practical validation in STX:LIBJAVA.

The paper is organized as follows: Section 2 discusses integration problems in detail. Section 3 gives an overview of STX:LIBJAVA and its implementation. Sections 4, 5, 6 and 7 describe techniques to solve these problems. Section 8 presents some real-world examples to validate our solution. Section 9 discusses related work. Finally, Section 10 concludes this paper.

2. Problem Description

At first glance Smalltalk and Java are very similar. Both are high-level object oriented, class-based languages with single inheritance and automatic memory management. In both languages message sending is the fundamental way of communication between objects. In this section we enumerate details by which these languages differ and which pose problems when integrating these languages into a common system.

1. **Class access.** In Smalltalk classes are identified by name. There is only one class with a given name at any time (although it may change over time) and it must be present and resolved prior to the code actually beeing executed. The system triggers a runtime error otherwise. The process of loading classes into the system is not specified in the standard, although some Smalltalk dialects provide namespaces and/or a lazy class loading facility.

   On the other hand Java provides a well-defined, user-extensible mechanism called *classloaders* for lazy-loading of classes into a running system. Moreover in Java a class is not only identified by its name but by its defining classloader as well. In other words two possibly different classes with the same name may coexist in the running system as long as they have been defined by different classloaders. Which classloader is used to load a particular class at a particular place in the code is a subject to complex rules and depends on the runtime context.

2. **Selector mismatch.** On the bytecode level, a method is identified by a *selector* in both languages. However, the syntactic format of selectors differs. Smalltalk uses the same selector in bytecode and in a source code. Java encodes type information into a selector at the bytecode level. This may affect reflection and/or dynamic execution, eg via `#perform:`.

   For example, consider the following code in Smalltalk:

   ```
   out println: 10
   ```

   Let's assume that `out` is an instance of the Java class `java.io.PrintStream`. Smalltalk compiles a message send with the selector `println:`. However, at the bytecode level the Java selector is `println(I)V`, whereas `println` is the method name expressed in the source code. The added suffix `(I)V` means that the method takes one argument of the Java type `int` and `V` means that the method does not return a value (aka returns a void type).

3. **Method overloading.** Java has a concept of overloading: in a single class, multiple methods with same name but with different numbers or types of arguments may coexist. For example, a `PrintStream` instance from the previous example has multiple methods by the name `println`, one taking an argument of type `int`, another taking a Java `String` argument, and so on. On the virtual machine level, each overloaded method has a different selector (`println(I)V`, respectively `println (Ljava/lang/String;)V`).

   The method which is called depends on the static types of the argument types at the call site. At compile time the Java compiler finds the best match and generates a send with the particular selector. For example, the source code

```
out.println("String");
out.println(1);
```

produces the following bytecode with type information encoded in method selectors (arguments of `INVOKEVIRT` instruction):

```
ALOAD ...
LDC 1
INVOKEVIRT println(L...String;)V
BIPUSH 1
INVOKEVIRT println(I)V
```

As Smalltalk is dynamically typed, no type specific selector are chosen by the Smalltalk compiler at compile time.

4. **Protocol mismatch.** Comparing Java and Smalltalk side by side, many classes with similar functionality are found. For example both Java's `java.lang.String` and Smalltalk's `String` class represent a `String` type. More complex examples are `java.util.Map` and Smalltalk's `Dictionary` class. Here, the classes have different names, possibly have different method names, but their purpose and usage is similar — both store objects under a particular key.

   When using code from both languages the programmer has to take care which kind of object (Java or Smalltalk) is passed as an argument. For example the hash of a string is obtained by sending the `hashCode` message in Java but `hash` in Smalltalk. Passing a Smalltalk String as an argument to a Java method which expects a Java-like String may result in a `MethodNotFound` exception.

5. **Exceptions.** The exception mechanisms in Smalltalk and Java differ in two very profound ways:

   First in Smalltalk both exception and ensure-handlers are blocks. Blocks are self-contained first-class closures that can be manipulated and executed separately. In Java exception and finally-handlers are syntactic entities and technically generate a sequence of bytecode instructions, which are spliced into the instruction stream. They can be only executed by jumping to the beginning of the handler and then continuing execution from that location. The method in question contains a special *exception table* which maps instruction ranges to the beginning of a particular handler.

   Second, Smalltalk provides resumable exceptions. When an exception is thrown (raised) in Smalltalk, the handler is executed on top of the context that caused that exception. The underlying stack frames are still present at handler execution time. If the handler returns, *i.e.,* the exception is not "proceeded"[1], ensure blocks are executed after the exception handler. Java provides only return semantics. When a handler is found, all contexts up to the handler context are immediately unwound and the execution is resumed at the beginning of the handler. Any finally block is treated like special exception handler, which matches any exception.

   In other words, the main difference between Smalltalk and Java exceptions from the programmer's point of view is that in Smalltalk, ensure blocks are *eventually* executed *after* the the handler, if the handler decides to return. In contrast, finally blocks of Java are *always* executed and their evaluation happens *before* the execution of the handler. A Smalltalk handler is even free to dynamically decide whether to proceed with execution after the exception is raised. Such proceedable exceptions are useful to continue execution after fixing a problem in the handler, or to implement queries or notifications (for example, to implement loggers or user interaction).

6. **Synchronization.** The mechanisms for process synchronisation differ both in design and implementation.

   The principal synchronisation mechanism in Java is a *monitor*. Conceptionally every Java object has associated with it a monitor object. Synchronization is done using two basic operations: *entering* the monitor, which may imply a wait on its availablility and *leaving* the monitor (Section 8.13, Lindholm and Yellin [10]). Monitor *enter* and *leave* operations must be properly nested. In Java, whole methods can be marked as *synchronized*, in which case the Java virtual machine itself is responsible for *entering* the monitor associated with the receiver and *leaving* it when the method returns. This happens both for normal and unexpected returns, for example due to unwinding after an uncaught exception. For *synchronized blocks*, the compiler emits `MONITORENTER` and `MONITOREXIT` bytecodes and ensures that no monitor remains entered, no matter how the method returns. Again, both normal and unwinding (Section 8.4.3.6, Gosling et al. [8], Section 3.11.11, Lindholm and Yellin [10]). The compiler achieves this by generating special finally-handlers, which *leave* the monitor and rethrow the exception.

   In Smalltalk (Smalltalk/X, in particular), processes are usually synchronized by semaphores. The programmer is responsible for proper semaphore signaling, although library routines provide support for critical regions. Technically speaking, there is no support at the virtual machine level for semaphores, except for a few low-level primitive methods.

   Now consider the following code:

```
[
  self basicExecute
] on: ExecutionError do: [ :ex |
```

[1] Smalltalk allows for exceptions to be "proceeded", which effectively means that the executions is resumed at the point where the exception was thrown (raised).

```
  self handleError: ex
]
```

A system which integrates Java and Smalltalk must ensure that all monitors possibly entered during execution of the `basicExecute` method are left when an `ExecutionError` is thrown and caught by the handler.

3. STX:LIBJAVA

3.1 In a Nutshell

STX:LIBJAVA is an implementation of the Java virtual machine built into the Smalltalk/X environment. In addition to providing the infrastructure to load and execute Java code, it also integrates Java into the Smalltalk development environment, including browsers, debugger and other tools.

Calling Java from Smalltalk is almost natural. Listing 1 demonstrates how a Java library can be used from Smalltalk. In the example, an XML file is parsed and parsed data is printed on a system transcript. The XML file is parsed by the SAX parser, which is completely implemented in Java. However, the SAX events are processed by a Smalltalk object — an instance of CDDatabaseHandler (see Listing 2).

This example demonstrates STX:LIBJAVA interoperability features:

- Java classes are referred to via a sequence of (unary) messages, which comprise the fully qualified name of a class. Java classes can be reached via the global variable `JAVA`. For example, to access `java.io.File` from Smalltalk, one may use[2]:

  ```
  JAVA java io File
  ```

- To overcome selector mismatch errors, STX:LIBJAVA intercepts the first message send and automatically creates a *dynamic proxy method*. These dynamic proxy methods translate the selector from Smalltalk keyword form into a correctly typed Java method descriptor and pass control to the corresponding Java method.
- STX:LIBJAVA provides a bridge between Java and Smalltalk exception models. Java exceptions can be handled by Smalltalk code. When a Smalltalk exception is thrown and the handler returns, all Java finally blocks between the raising context and handler context are correctly executed and all possibly entered monitors are left. No stale Java locks are left behind.

3.2 Architecture of STX:LIBJAVA

In this section we will briefly outline STX:LIBJAVA's internal architecture.

Unlike other projects which integrate Java with other languages, STX:LIBJAVA does not use the original JVM in parallel with the host virtual machine, nor does it translate Java source code or Java bytecode to any other host language. Instead the Smalltalk/X virtual machine is extended to support multiple bytecode sets and execute Java bytecode directly. To our knowledge, Smalltalk/X and STX:LIBJAVA is the only programming environment that took this approach.

The required infrastructure for loading `.class` (chapter 4, Lindholm and Yellin [10]) files, class loader support and additional support for execution, such as native methods, is implemented in Smalltalk. Java runtime classes and methods are implemented as customized Smalltalk `Behavior` and `Method` objects. In particular, Java methods are represented as instances of subclasses of the Smalltalk *Method* class. However, they refer to Java instead of Smalltalk bytecode. Execution of Java bytecode is implemented in the virtual machine. In the same way that Smalltalk bytecode is handled by the VM, Java bytecode is interpreted and/or dynamically compiled to machine code (jitted).

However, some complex instructions (such as `CHECKCAST` or `MONITORENTER`) are handled by the virtual machine calling back into the Smalltalk layer via a so-called trampoline and are implemented in Smalltalk as a library method. Similarly, all native methods are implemented in Smalltalk.

Both Smalltalk and Java objects live in the same object memory and are handled by the same object engine and garbage collector. Performance-wise, there is no difference between Smalltalk code calling a Java method or other Smalltalk code. Moreover, all dynamic features of the Smalltalk environment - such as stack reification and advanced reflection — can be used on the Java code.

The main disadvantage of our approach (as opposed to having a separate original JVM execute Java bytecodes) is that the whole functionality of the Java virtual machine. has to be reimplemented. This includes an extensive number of native methods, which indeed involve a lot of engineering work. However, we believe that this solution opens possibilities to a much tighter integration which would not be possible otherwise.

4. Class Access

In Smalltalk classes are stored by name in the global Smalltalk dictionary. Obviously, this dictionary cannot be reused by the Java subsystem, as Java classes are specified by name and defining class loader. Therefore loaded classes are accessed through `JavaVM` class. Listing 3 shows its usage in case of a known class loader instance.

```
JavaVM
  classForName: 'org.junit.TestCase'
  definedBy: classLoaderObject
```

Listing 3. Accessing Java class with known class loader

[2] alternatively, the class can also be referred to via a sub-namespace, as `JAVA::java::io::File`.

```
factory := JAVA javax xml parsers SAXParserFactory newInstance.
parser := factory newSAXParser getXMLReader.
parser setContentHandler: JavaExamples::CDDatabaseHandler new.
[
  parser parse: 'cd.xml'.
] on: JAVA java io IOException do:[:ioe|
  Transcript showCR: 'I/O␣error:␣', ioe getMessage.
  ioe printStackTrace
] on: UserNotification do:[:un|
  Transcript showCR: un messageText.
  un proceed.
]
```

Listing 1. Smalltalk code calling Java XML parser

```
CDDatabaseHandler>>startElement:namespace localName:localName qName:qName attributes:
    attributes
  tag := qName.

CDDatabaseHandler>>endElement:namespace localName:localName qName:qName
  qName = 'cd' ifTrue:[
    title isNil ifTrue:[self error: 'No␣title'].
    artist isNil ifTrue:[self error: 'No␣artist'].
    index := index + 1.
    UserNotification notify:
      (index printString , '.␣', title , '␣-␣' , artist)
  ]

CDDatabaseHandler>>characters: string offset: off length: len
  tag = 'title'  ifTrue:[
    title := string copyFrom: off + 1 to: off + len.
    tag := nil.
  ].
  tag = 'artist' ifTrue:[
    artist := string copyFrom: off + 1 to: off + len.
    tag := nil.
  ].
```

Listing 2. An excerpt of `CDDatabaseParser` used in Listing 1

As already shown in Listing 1, the `JAVA` global variable is provided to refer to a Java class using the current class loader.

Interoperability approaches based on foreign-function interfaces of the JVM and host virtual machine suffer from the inability to reclaim classes which have entered the JNI (Java native interface). In STX:LIBJAVA all loaded classes are reclaimed in compliance with the JVM specification (Section 12.7, Gosling et al. [8]).

5. Dynamic Proxy Methods

The *Dynamic Proxy Method* is a mechanism employed by STX:LIBJAVA to solve selector and protocol mismatch and to deal with Java's method overloading. A proxy method is an intermediate method that possibly performs an additional method resolution, transforms arguments and finally passes control to a real method dispatching on the type and number of arguments. Such a proxy is generated dynamically whenever the control flow crosses the language boundary, *i.e.,* when Smalltalk calls Java or vice versa. In the following sections we describe in detail how dynamic method proxies solve the problem outlined in the previous sections. We will demonstrate proxies on examples of Smalltalk calling Java; the actions performed in the opposite call direction are analogous.

5.1 Selector Mismatch

Consider the example given in the listing 4. Without additional interoperability support, a `DoesNotUnderstand` exception would be raised, since there is obviously no method for the `println:` selector in the Java `PrintStream` class.

```
out := JAVA java lang System out.
out println: 10.
```

Listing 4. Example of selector mismatch

STX:LIBJAVA's interoperability mechanism catches cross-language message sends and dynamically generates a *dynamic proxy method* for the original selector, which performs a second send using a transformed selector. The code of the proxy is shown on Listing 5. The proxy is compiled on the fly and installed into the receiver's class and the original message-send is restarted. This way a proxy method is generated only for the very first time and subsequent sends will use the "fast path", invoking the already generated proxy directly. Details on how sends are intercepted are discussed below in section 5.6.

```
java.io.PrintStream>>println: arg
  self perform: #'println(I)V'
          with: arg
```

Listing 5. A proxy method for `println()` Java method

5.2 Method Resolution

There are numerous ways to translate Smalltalk selectors to corresponding Java selectors and vice versa. This section does not discuss any pros and cons of possible approaches. We do not believe that there is the only one the best way how to translate selectors. STX:LIBJAVA simply uses the way which showed to be the most natural and easy for our purpose. The translation rules are described below.

Smalltalk to Java resolution. When calling a selector like `println:`, and the receiver is a Java object, the Java object and its super-classes are searched for all methods with a name of `println`. In case there is no such method, the `#doesNotUnderstand:` message is sent, as usual. If exactly one method exists by that name, that method is invoked. In case more than one method exists by the name (*e.g.,* the method is overloaded), the algorithm consults at run-time the number and types of arguments and tries to find the best matching Java method. If the number of arguments does not match, the `#doesNotUnderstand:` message is sent. Finally, if an interface type is expected as an argument and the argument is a Smalltalk object (which usually does not implement the Java interface), STX:LIBJAVA follows the traditional Smalltalk duck-typing philosophy and passes the Smalltalk argument unchanged to the method. Either the argument object implements any required interface methods (and everything works as expected then), or Java throws a runtime exception, which can be handled either by Smalltalk or by Java code.

Java to Smalltalk resolution. When calling a selector like `put(Ljava.lang.String;Ljava.lang.Object)V`[3]. and the receiver is a Smalltalk object, the object's class and its superclasses are searched for any selector starting with the first keyword part, `put:`. The rest of the selector is ignored in this matching process. However, the number of arguments must match. Also argument types are ignored. If more than one method fulfils these criteria, an `AmbiguousMessageSend` error is raised.

Variable number of arguments. Starting with Java 1.6, Java supports a variable number of arguments (section 8.4.1, Gosling et al. [8]). Technically, variable arguments are wrapped into an array object and passed to the method as a single argument. A Java compiler is responsible for generating code that wraps variable arguments. Therefore, no special care is required during method resolution.

5.3 Method Overloading

To demonstrate method overloading, we extend the example in Listing 4 , as depicted in Listing 6. After execution of line 3, a new proxy method has been added to the `java.io.PrintStream` class as depicted in Listing 5. The execution of line 4 would raise a runtime error, since we call the `println(I)V` method with a `boolean` parameter.

```
1 out := JAVA java lang System out.
2 out println: 10.
3 out println: true.
```

Listing 6. An example of overloaded method called from Smalltalk

```
java.io.PrintStream>>println: a1
  | method |
  (a1 class == SmallInteger) ifTrue:[
    ↑ self perform: #'println(I)V' with:
        a1
  ].
  self recompile: a1.
  ↑ self println: a1.
```

Listing 7. A proxy method for an overloaded method

Due to the dynamic nature of Smalltalk, argument types cannot be statically inferred and may even change during execution. Therefore, another method resolution step has to be added to the proxy method. To ensure type-safety (as

[3] A Java selector for method named `put` that takes two arguments of types `java.lang.String` and `java.lang.Object` respectively and whose return type is `void`

required and assumed by the Java code) the actual call to the Java method is protected by a "guard" that checks the actual argument types. The code for the `println:` proxy after execution of line 2 is shown in Listing 7.

Only one guard is added at a time. Line 6 ensures that if no guard matches — like when line 3 of example from Listing 6 is executed — the proxy is recompiled, possibly adding a new guard. Line 7 restarts the send. This prevents unnecessary guards which are actually never used from being generated. An alternative implementation based on multiple dispatch could be implemented by dynamically installing double dispatch methods in the encountered argument classes. However, as the number of dynamically encountered argument types is usually relatively small, we believe that the switch code on the argument class is usually sufficient and faster.

5.4 Protocol Mismatch

In some cases, it is not sufficient to simply translate the Java selector to a Smalltalk selector and vice versa. For example, Smalltalk code can send a `#isEmpty` message to a `java.lang.String` (because it contains the `isEmpty()` method), but it cannot send `#collectAll:`, as there is no such functionality in a Java string. Therefore the arguments and return values should also be converted when crossing the language boundary. STX:LIBJAVA makes these conversions automatically for predefined types (such as String, Integer, Boolean, ...). Thanks to dynamic proxy methods, user-defined types can be converted automatically as well.

Being aware of the protocol mismatch problem, we have to update the proxy method from Listing 7 as depicted in Listing 8. The `#asJavaObject` and `#asSmalltalkObject` are responsible for conversion between Smalltalk and Java types.

```
java.io.PrintStream>>println: a1
  |  jA1 jA1Class |
  jA1 := a1 asJavaObject.
  jA1Class :=
    Java classForName:
      'java/Lang/String'.

  (jA1 class == jA1Class) ifTrue: [
    ↑ (self #'println(Ljava/lang/String;)
        V': jA1)
          asSmalltalkObject.
  ].
  self recompile: a1.
  ↑ self println: a1.
```

Listing 8. A proxy method with argument and return value conversions

5.5 Field Accessing

In Java, public fields can be accessed directly using the dot-notation whereas in Smalltalk, values of instance variables could only be accessed using accessor methods. Although in modern Java, declaring instance fields `public` and accessing them directly is considered as a "bad style", public static fields are often thorough Java libraries to expose constant values. STX:LIBJAVA allows for public field to be accessed from Smalltalk in a Smalltalk way,*i.e.,* by access methods. These accessor methods are dynamically compiled whenever needed - just like proxy methods described above.

Consider the example given in the listing 9 that access public static field `PI` of class `java.lang.Math`. STX:LIBJAVA interoperability mechanism catches the send of message `#PI` to the class and automatically generates getter method returning corresponding field.

```
cf = 2 * (JAVA java lang Math PI) * r
```

Listing 9. An example of accessing Java fields from Smalltalk

Accessor methods are generated only for Java fields declared as `public`. If the field is declared as `final`, only getter method is generated. Same mechanism is used to access both static and instance fields.

5.6 Intercepting the Message Send

To install a proxy, a message send must be intercepted. A standard Smalltalk solution would be to override the `#doesNotUnderstand:` method so it creates and installs the generated proxy method.

However, STX:LIBJAVA utilizes the method lookup meta-object protocol (MOP; Vraný et al. [12]) which is integrated into the Smalltalk/X virtual machine. The MOP allows for a user-defined method lookup routine to be specified on a per-class basis. This user-defined method lookup routine installs the proxy and invokes it.

6. Mixed Exception-Handling

Considering the mixed Smalltalk and Java code in Figure 1 (part a) which creates a user account, let us follow the actions taken when an exception is thrown in an invocation of `CreateAccountCmd»perform` (part b). First, the handler is searched and executed (Step 1 in Figure 1). Since the handler does not proceed the stack should be unwound and control passed to the `createAccountClicked` method. All ensure and finally blocks between the current context and the context for `createAccountClicked` method are executed first. The first such block is the ensure block in the `execute` method (Step 2 in Figure 1). The second is the finally block in `createAccount()`. However, the finally block is not a Smalltalk block and thus cannot be evaluated easily. Due to the underlying virtual machine implementation, the only way to execute the finally code is to (i) set the program counter to the beginning of the handler and (ii) restart the method. Restarting a method also implies that contexts below the restarted method's context are destroyed (*i.e.,* contexts for called methods). In particular, the context of the `raise` method should be destroyed. This poses a problem, as it is the context of the method which controls all handler and ensure block evaluation.

To solve this problem we exploit two facts. First, when ensure or finally handlers are executed, the contexts below the handler context are going to be destroyed anyway. Second, finally blocks are compiled in such a way that they never touch the exception object. The exception is only temporarily stored and then rethrown by the `ATHROW` Java instruction. The exception object can be any object, not necessarily a Java object inheriting from `Throwable`. To execute the finally code, we pass a special *finally token* object as an exception and restart the method with the finally block (Step 3 in Figure 1). This special *finally token* is recognised by the `ATHROW` instruction. Upon detection of that special token, `ATHROW` continues to evaluate finally and ensure blocks and finally unwinds the stack (Step 4 in Figure 1).

7. Synchronization

The Java runtime library has been designed to be thread-safe. Critical sections guarded by monitors are pervasive through the code. Correct handling of monitors in case of mixed Java-Smalltalk code is essential as a single leftover monitor can easily result in blocking the whole system.

Prior to entering a critical section, a monitor is *entered*, which must be *left* at the end of the section. This is done either implicitly by the virtual machine (when the whole method is marked as *synchronized*) or explicitly using special `MONITORENTER` and `MONITORLEAVE` instructions. Those implement fine-grain synchronization of Java's *synchronized blocks*.

7.1 Synchronized blocks

When an exception occurs during the execution of a synchronized block, the guarding monitor must be left before the control flow is passed to a handler higher up on an execution stack [4]. To ensure this, a Java compiler must generate a synthetic finally block in which the monitor is left and the exception is rethrown (Section 7.13, Lindholm and Yellin [10]). Figure 2 shows an example of such a finally block.

Consider the code shown in Figure 2. Because an exception could be thrown in the synchronized block (bytecode lines 2 - 10), the compiler generates a special handler (bytecode lines 11 - 15, Figure 2), in which the monitor is left and the exception is rethrown.

7.2 Synchronized methods

In the case of synchronized methods, neither `MONITOR-ENTER` / `MONITOREXIT` instructions, nor synthetic finally blocks are generated by the Java compiler. Instead, synchronization is done directly by the virtual machine. The monitor used for synchronization is the monitor associated with the receiver or (in case of a static method) with the class.

Prior to executing a synchronized method, the virtual machine *enters* the monitor associated with the receiver and tags the context as an *unwind context*. The VM intercepts returns through such tagged contexts and trampolines into the Smalltalk level for any unwind actions to be executed. In this case, the monitor-leave semantic is performed.

8. STX:LIBJAVA at Work

As mentioned in the 1 section, language integration consists of runtime-level and language-level integration.

Runtime-level integration. Several reasonably large Java projects have been chosen as benchmarks to validate the correct implementation of Java runtime support. These projects include:

- Apache Tomcat[5] – a Servlet/JSP container,
- SAXON[6] – an XSLT and XQuery processor,
- Groovy[7] – a dynamic language for Java platform,

Apache Tomcat makes heavy use of almost all Java features, from threads and synchronization, through dynamic class generation, class loading and finalization, to exceptions and finally blocks. Groovy makes heavy use of Java reflection and especially class loading as it dynamically generates Java bytecode and loads it into a running system. All these programs run correctly under STX:LIBJAVA.

Language-level integration. Language-level interoperability was already demonstrated in Listings 1 and 2, which use the Xerces XML parser to parse an XML file. The filename is passed into the Java method as a Smalltalk string. The SAX handler which is passed to the parser and later called for decoded elements is actually implemented in Smalltalk. No boilerplate code is needed whatsoever.

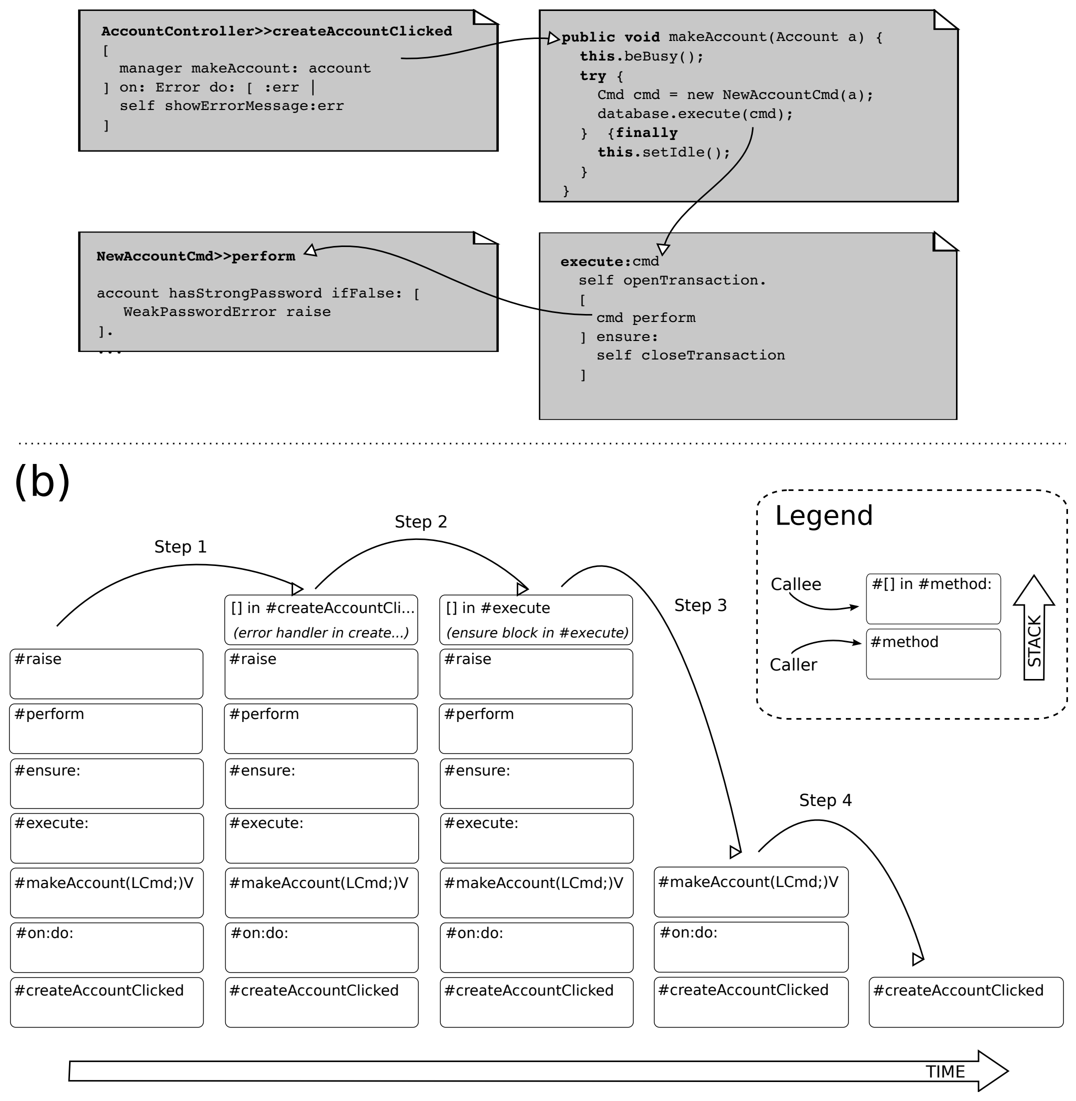
Tool support. A Java implementation in Smalltalk would not be complete without support in the development tools. Java classes can be browsed using the standard Smalltalk class browser, and Java objects or classes can be inspected in the Smalltalk inspectors. For programmer convenience, specialized inspectors are provided for specific Java classes such as `Vector`, `ArrayList`, `Set` or `Map`. Java is also fully supported by the debugger — breakpoints may be set on methods and Java code can be single stepped and debugged just like Smalltalk. A Groovy interpreter has been integrated into the Workspace application so programmers

[4] assuming that the stack grows downwards

[5] http://tomcat.apache.org

[6] http://saxon.sourceforge.com

[7] http://groovy.codehaus.com



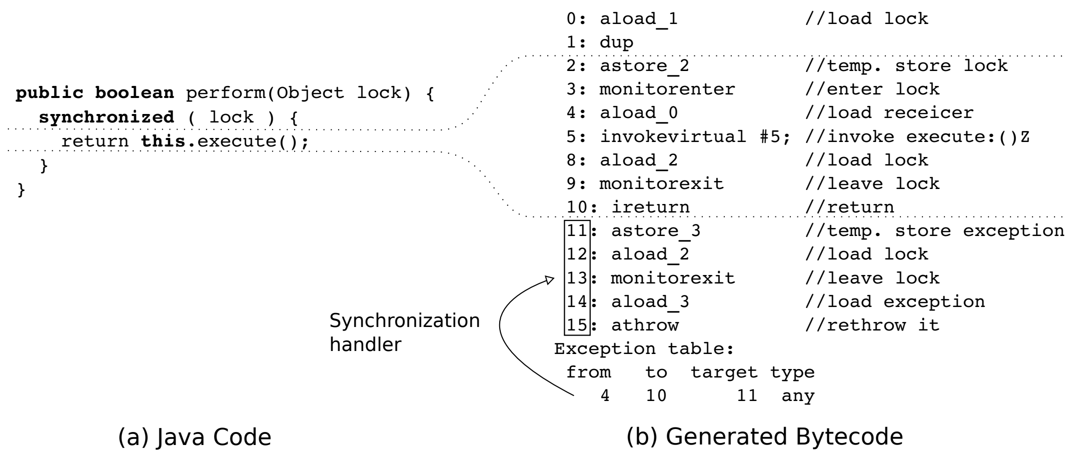
```
AccountController>>createAccountClicked
[
  manager makeAccount: account
] on: Error do: [ :err |
  self showErrorMessage:err
]
```

```
public void makeAccount(Account a) {
  this.beBusy();
  try {
    Cmd cmd = new NewAccountCmd(a);
    database.execute(cmd);
  }  {finally
    this.setIdle();
  }
}
```

```
NewAccountCmd>>perform

account hasStrongPassword ifFalse: [
   WeakPasswordError raise
].
...
```

```
execute:cmd
  self openTransaction.
  [
    cmd perform
  ] ensure:
    self closeTransaction
  ]
```

Figure 1. (a) an example of mixed-exception code (b) method activation stackswhen executing code from (a). Too keep the figure concise, unimportant intermediate contexts for "try" blocks in `#on:error:` and `#ensure:` are omitted.



```
public boolean perform(Object lock) {
  synchronized ( lock ) {
    return this.execute();
  }
}
```

```
 0: aload_1           //load lock
 1: dup
 2: astore_2          //temp. store lock
 3: monitorenter      //enter lock
 4: aload_0           //load receicer
 5: invokevirtual #5; //invoke execute:()Z
 8: aload_2           //load lock
 9: monitorexit       //leave lock
10: ireturn           //return
11: astore_3          //temp. store exception
12: aload_2           //load lock
13: monitorexit       //leave lock
14: aload_3           //load exception
15: athrow            //rethrow it
Exception table:
 from   to  target type
   4    10      11  any
```

Figure 2. Example of synchronized block in Java

can quickly test Java code (doIt), in just the way they are used to with Smalltalk. Also, JUnit[8] has been integrated into the standard tools to ease running of JUnit tests.

Java classes can be unloaded and then a possibly new version can be loaded again at run-time – a feature missing in other Java virtual machines. A new can be even written and then "accepted" in the class browser. A standard `javac` is then invoked a new class is reloaded into a running system. However, this feature is still experimental.

9. Related Work

9.1 JavaConnect and JNIPort

JavaConnect (Brichau and De Roover [1]) and **JNIPort** (Geidel [4]) are Smalltalk libraries that allow interaction with Java code from within Smalltalk. A Java virtual machine is linked into the Smalltalk virtual machine and the communication is made via foreign-function interfaces. Cross-language messages are translated into FFI invocations of either environment.

Java classes can be browsed, but cannot be modified from within Smalltalk. This is caused by the implementation of the Java virtual machine. Each Java object passed through FFI must be wrapped and registered, which generates additional overhead and disables automatic garbage collection of such objects. Because Java code is running in a separate virtual machine, proxy Java class must be generated for every Smalltalk class passed into Java. To reduce the overhead due to FFI calls, JavaConnect introduces the concept of language shifting objects. Shifted Java objects have part (or whole) of their behavior translated into Smalltalk, but not all instructions and language constructs are supported (such as `MONITORENTER` instruction and `synchronized` methods). In multithreaded applications, object state must be synchronized and problems arise, when a single native-threaded Smalltalk virtual machine interacts with a multithreaded Java application. Deadlocks can occur when a Java thread tries to communicate with the Smalltalk virtual machine, whose single native thread is blocked by another Java thread.

In STX:LIBJAVA, Java classes can be created, modified, or destroyed in runtime. There is no need to synchronize object state across two virtual machines. Java methods are directly executed, therefore no translation or interprocess communication is needed.

9.2 IBM VisualAge

IBM VisualAge for Java (Deupree and Weitzel [2]) includes an interaction mechanism on the Smalltalk side. Communication is realized using remote-method invocation (RMI). Java and Smalltalk virtual machines run in parallel, possibly even on two separate machines. The transition between languages is explicit and managed by the programmer. A lot of boilerplate code must be written and objects must be registered, converted and maintained by the programmer when crossing language barrier.

STX:LIBJAVA does not require any boilerplate code to access code across the language barrier, nor does it require any explicit handling or conversion when crossing the barrier. There is no need to set up an RMI service and to explicitly register objects in the RMI registry.

9.3 Redline Smalltalk

Redline Smalltalk (Ladd [9]) is a Smalltalk implementation running on the JVM. Java classes are dynamically compiled from Smalltalk source code and loaded into JVM using standard class loaders. Smalltalk exceptions are mapped onto Java exceptions. Therefore, it is not possible to proceed or retry an exception. This greatly restricts the number of Smalltalk applications and libraries which could run on Redline Smalltalk. Some methods, especially those related to object space reflection, such as `allInstances` or `become:`, are not supported for performance reasons. Common types such as string and integer must be explicitly converted when crossing the language barrier. Due to Java's static type system, Smalltalk objects can only be passed as argument if the Java method expects that type.

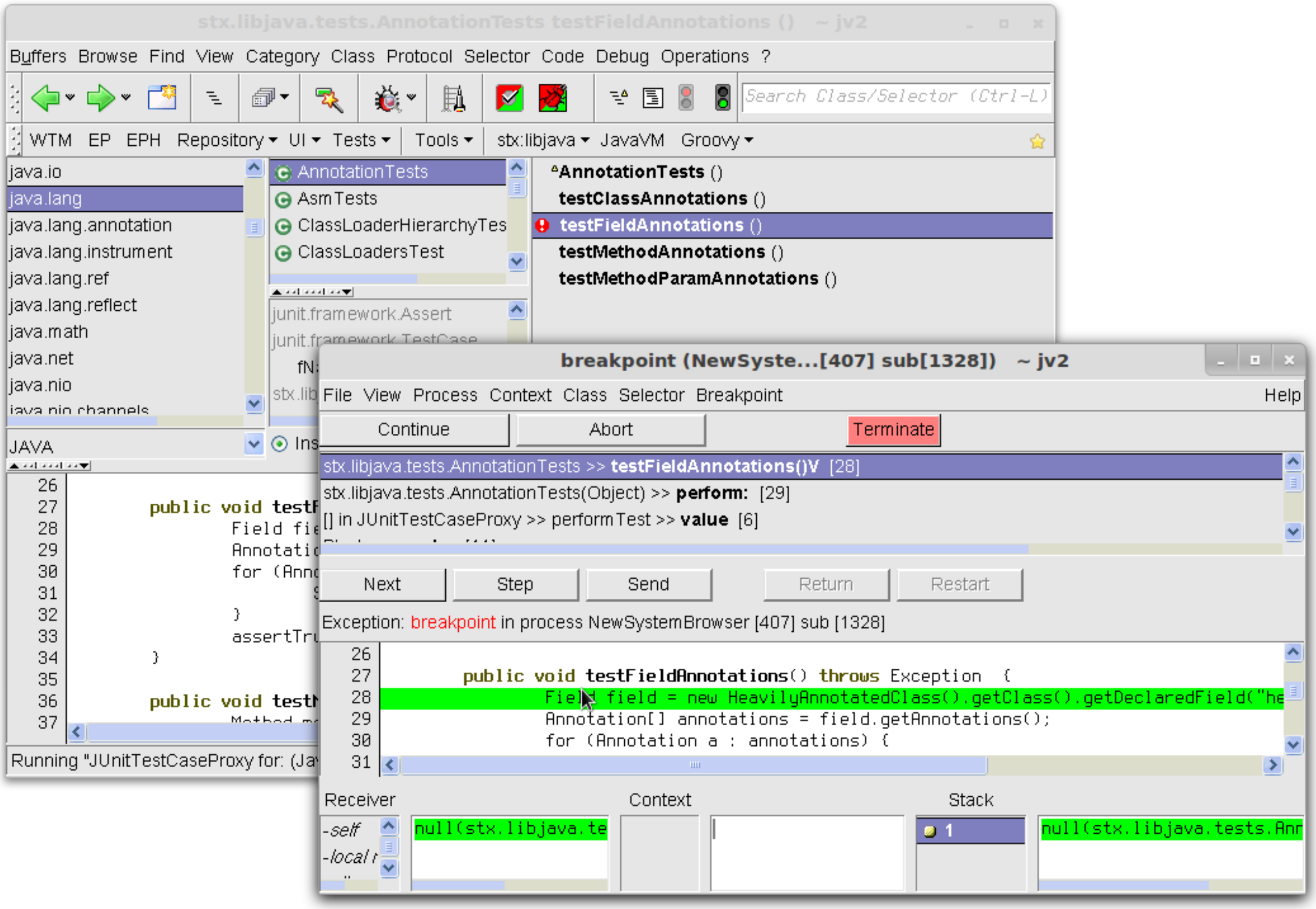
STX:LIBJAVA does not alter any feature of the Java language, Java reflection is also fully compliant with the original JVM. Any Java application could run on STX:LIBJAVA without modifications. Common types are automatically converted. Java and Smalltalk methods can be passed in as needed and the programmer is freed from the need to handle Java or Smalltalk objects differently. STX:LIBJAVA is currently the only language implementation of this kind on Smalltalk/X, however the approach does allow for direct communication between other such languages if they become relevant[9].

10. Conclusion And Future Work

In this paper, we have presented STX:LIBJAVA, a Java virtual machine implementation integrated into the Smalltalk/X environment. We have described the main problems of a seamless integration of Smalltalk and Java languages and their solutions. We use dynamic method proxies to allow for Java code to be easily called from Smalltalk and the other way round. Also, we described how to integrate Smalltalk and Java exception and synchronization mechanisms, so Smalltalk code can handle Java exceptions while the semantics of Java finally and synchronized blocks are preserved. A number of significantly large programs such as SAXON XSLT processor and Apache Tomcat Server/JSP container run on STX:LIBJAVA.

[8] www.junit,org

[9] Ruby (Vraný [11], Chapter 8) and a JavaScript dialect (Gittinger [5]) have been integrated into are Smalltalk/X, but these translate the source language into Smalltalk/X bytecode and do not suffer from the complexity resulting from major semantic differences of eg. the exception mechanism



```
        public void testF
                Field fie
                Annotati
                for (Ann
                        s
                }
                assertTr
        }

        public void testM
                Method m
```

Running "JUnitTestCaseProxy for: (Ja

stx.libjava.tests.AnnotationTests >> testFieldAnnotations()V [28]
stx.libjava.tests.AnnotationTests(Object) >> perform: [29]
[] in JUnitTestCaseProxy >> performTest >> value [6]

Exception: breakpoint in process NewSystemBrowser [407] sub [1328]

```
    public void testFieldAnnotations() throws Exception  {
        Field field = new HeavilyAnnotatedClass().getClass().getDeclaredField("he
        Annotation[] annotations = field.getAnnotations();
        for (Annotation a : annotations) {
```

```
null(stx.libjava.te
null(stx.libjava.tests.Anr
```

Figure 3. Class Browser and Debugger showing Java code

In the future we plan to further improve the integration of Java into Smalltalk environment, for instance: add support for extension methods on Java classes, integrate Java support to Smalltalk/X packaging tools and building process, improve code highlighting and navigation, add more specialised object inspectors. We also plan to extend STX:LIBJAVA to provide a fully incremental development environment for Java, similar to that provided for the Smalltalk language.

Acknowledgement. We would like to gratefully thank to Oscar Nierstrasz and Eliot Miranda for their precious comments.

References

[1] J. Brichau and C. De Roover. Language-shifting objects from java to smalltalk: an exploration using javaconnect. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '09, pages 120–125, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-899-5. doi: 10.1145/1735935.1735956. URL http://doi.acm.org/10.1145/1735935.1735956.

[2] J. Deupree and M. Weitzel. Visualage integration for the 21st century: Smalltalk, java, websphere. URL http://www-01.ibm.com/support/docview.wss?uid=swg27000174.

[3] Expecco. Java interface library 2.1. http://wiki.expecco.de/wiki/Java_Interface_Library_2.1.

[4] J. Geidel. Jniport, Feb. 2011. URL http://jniport.wikispaces.com/.

[5] C. Gittinger. Javascript compiler and interpreter. http://live.exept.de/doc/online/english/programming/goody_javaScript.html.

[6] C. Gittinger. Die Unified Smalltalk/Java Virtual Machine in Smalltalk/X. *In Proceedings of NetObjectDays*, 1997.

[7] C. Gittinger and S. Vogel. Smalltalk/Java Integration in Smalltalk/X. *Tagungsband STJA'97, GI Fachtagung Objektoriente Softwareentwicklung*, 1997.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, The (3rd Edition)*. Addison Wesley, Santa Clara, California 95054, U.S.A, 3 edition, 6 2005. ISBN 9780321246783. URL http://java.sun.com/docs/books/jls/.

[9] J. Ladd. Smalltalk implementation for the jvm. URL www.redline.st.

[10] T. Lindholm and F. Yellin. *Java^{TM} Virtual Machine Specification, The (2nd Edition)*. Prentice Hall, Santa Clara, California 95054 U.S.A, 2 edition, 4 1999. ISBN 9780201432947. URL http://java.sun.com/docs/books/jvms/.

[11] J. Vraný. *Supporting Multiple Languages in Virtual Machines*. PhD thesis, Faculty of Information Technologies, Czech Technical University in Prague, Sept. 2010.

[12] J. Vraný, J. Kurš, and C. Gittinger. Efficient Method Lookup Customization for Smalltalk. *Objects, Models, Components, Patterns*, pages 1–16, 2012. doi: 10.1007/978-3-642-30561-0_10. URL http://www.springerlink.com/index/PU875371770R562R.pdf.

Smalltalk in a C World

David Chisnall

University of Cambridge

David.Chisnall@cl.cam.ac.uk

Abstract

Smalltalk, in spite of myriad advantages in terms of ease of development, has been largely eclipsed by lower-level languages like C, which has become the lingua franca on modern systems. This paper introduces the Pragmatic Smalltalk compiler, which provides a dialect of Smalltalk that is designed from the ground up for close interoperability with C libraries. We describe how high-level Smalltalk language features are lowered to allow execution without a virtual machine, allowing Smalltalk and C code to be mixed in the same program without hard boundaries between the two. This allows incremental deployment of Smalltalk code in programs and libraries along with heavily optimised low-level C and assembly routines for performance critical segments.

1. Introduction

Smalltalk was originally written for the Xerox Alto and closely integrated with the system. It ran directly on the hardware with the Smalltalk environment taking the place of an operating system. Describing this design, Dan Ingalls wrote 'The operating system is everything that doesn't fit in the language. It shouldn't exist'[17].

Modern Smalltalk VMs, such as Squeak and Pharo, often inherit this view, with a Smalltalk environment running inside, but isolated from, the rest of the system. This view is even adopted by Smalltalk-derived languages such as Java, which run in their own environment and have difficult interacting with foreign code. Any interactions with code outside of this world, for example native libraries, require a foreign function interface. This provides both a conceptual and a performance barrier and comes with composition problems if you want to mix code from two languages that each use the virtual machine model, both needing to go via a native-compatibility layer for interoperability.

International Workshop on Smalltalk Technologies August 28th, 2012, Gent
Copyright © 2012 ACM [to be supplied]. . . $10.00

The virtual machine approach arises from an historical curiosity: an implementation detail of how languages were implemented on the Xerox Alto. As described in [22], the Alto made heavy use of microcoding, with a low-level micro-instruction set shared by all languages and a higher-level instruction set for each target language. Each target language defined a bytecode, with each bytecode implemented as a short sequence of micro-instructions. Algol on the Alto, like Smalltalk, had a VM. This approach was important for the Alto because it was a research machine, and the use of microcoding made experimentation with the instruction set easier. The goal was to determine which instructions were useful in a CPU so that future CPUs could incorporate them. For example, the Smalltalk environment made use of a microcoded BitBLT instruction[16] to achieve good performance for a GUI, an operation that later appeared implemented entirely in hardware in early graphics accelerators. This goal is lost on modern VMs, where the VM is a vestigial indirection layer, retained because it - in theory, at least - allows code to run on multiple platforms.

Since the 1980s, Smalltalk has remained a niche language. In the same time, lower-level languages have accumulated large bodies of legacy code, much of which remains very useful. Low-level languages had an early advantage: although the cost of development is higher, the cost of the hardware needed to run software written in them was significantly lower. The larger potential market meant that significant amounts of effort were invested in development in these languages. For example, the Alto used to develop Smalltalk had 512KB of RAM in 1979, while the first Macintosh, shipping five years later and with a similar experience only required 128KB. Systems like the Apple II, which were contemporary with the Alto and were programmed mostly in assembly languages, typically cost over an order of magnitude less.

Interoperability is very important. At the time of writing, the amount of reusable C code that is publicly available dwarfs the amount of Smalltalk code similarly available. Ohloh.net, which collects statistics of public code repositories, counts 4,465,070,607 lines[1] of C code, plus 2,239,514,292 and 56,649,630 lines of C++ and Objective-C code respectively. In contrast, it only counts 2,041,366

[1] All figures were correct on 2012-06-08.

lines of Smalltalk code. This is a slightly misleading comparison, as it does not track code stored in Monticello, but even counting the amount of code in the public repositories it does not come close to the almost seven billion lines of [Objective-]C[++] code available and the greater expressiveness of Smalltalk does not bridge the gap.

This difference includes some omissions in key areas in Smalltalk, for example the lack of a modern HTML rendering engine and the relatively poor state of PDF rendering severely limit the possibilities of a pure Smalltalk environment. HTML and PDFs are two of the most popular formats for interoperability, and there are well-supported C/C++ libraries for displaying both, including embedded dynamic JavaScript elements. Modern video CODECs are similarly lacking Smalltalk implementations. This leads to an important choice: Should we aim to rewrite everything in Smalltalk, or to better interoperate with the large body of existing code? Even if we optimistically assume that a Smalltalk developer is 100 times more productive than a C developer, it is clear that the first approach would leave Smalltalk a long way behind.

2. Goals

The work described in this paper forms part of the Étoilé project, which, among other things, aims that no program should contain more than 1,000 lines of non-reusable code. All other code should be in frameworks and libraries that can be easily shared between applications. For example, a class encapsulating a window is a generic and reusable component and so does not count towards this total, whereas a class implementing a controller for a preference panel for a specific application would, as it is of no use to other applications.

To accomplish this goal, we need to make two things easy: reusing existing code and writing expressive descriptive new code. Smalltalk is a good fit for the second goal, as it provides a clean and elegant syntax for expressing high-level ideas. Existing implementations made this difficult, however.

Étoilé builds on the GNUstep project, which originally aimed to build an open source implementation of the OpenStep specification and now aims to track the extensions made in Apple's implementation of this specification: Cocoa. GNUstep, and its accompanying frameworks are written in Objective-C, which is a set of Smalltalk-like extensions to C, aimed at allowing C libraries to be reused via a loose-coupled object-oriented abstraction[13].

Our initial approach at combining languages was to implement bridges between GNU Smalltalk and Objective-C and between Io and Objective-C. This had numerous problems, including performance and the proliferation of proxies as the two (or three) languages all contained different underlying object models.

In this paper, we describe an implementation of Smalltalk that targets the same binary format as Objective-C, allowing Smalltalk and C to be mixed at the method granularity: a single object can have methods written in both languages. Within the scope of the overall goals of Étoilé, this implementation had the following goals:

- No virtual machine. There should be no run-time concept of 'Objective-C objects' versus 'Smalltalk objects', only objects that may have been written in either language.
- The ability to mix Objective-C and Smalltalk code at a fine granularity, with the message send as the boundary between the two worlds, not the object.
- The ability to easily use C++ code from Smalltalk (and vice versa).
- Support for native threading APIs and other low-level primitives with a very small overhead—syntactically or in terms of run-time cost—to encourage their use.
- Support for static compilation, to ease deployment.

An explicit non-goal of this project is a complete Smalltalk-80 environment. We (initially) aim to build a compiler and underlying framework that makes it possible to implement a Smalltalk-like environment, but we aim to use the OpenStep class libraries rather than the Smalltalk-80 ones. This is partly motivated by personal preference and partly because a large number of developers already have experience with these via Mac OS X and iOS.

We are not aiming to perform source-to-source translation implementing Smalltalk on top of Objective-C, we are aiming to implement both Objective-C, Smalltalk, and other languages on top of the same set of shared code. It would be possible to remove all of the Objective-C code from our implementation, if this is deemed desirable in the future, by rewriting it in Smalltalk.

This project is not limited to Smalltalk. We aim to provide a set of reusable components for implementing late-bound dynamic languages. As such, some of our improvements are usable in other contexts. For example, several of our optimisations also benefit Objective-C.

3. Design Overview

The implementation described in this paper is referred to as *Pragmatic Smalltalk*, and is built on top of—and developed as part of—the LanguageKit framework. LanguageKit is intended to be a reusable abstract syntax tree (AST), interpreter, JIT, and static compiler for dynamic object oriented languages. Smalltalk is the most actively used front end for LanguageKit, but there is also a proof-of-concept front end for a JavaScript-like language and an OMeta-like parser generator.

The compiler portions use LLVM[19], a set of libraries for writing optimising compilers. LLVM allows extra optimisations to be added relatively easily, transforming a low-

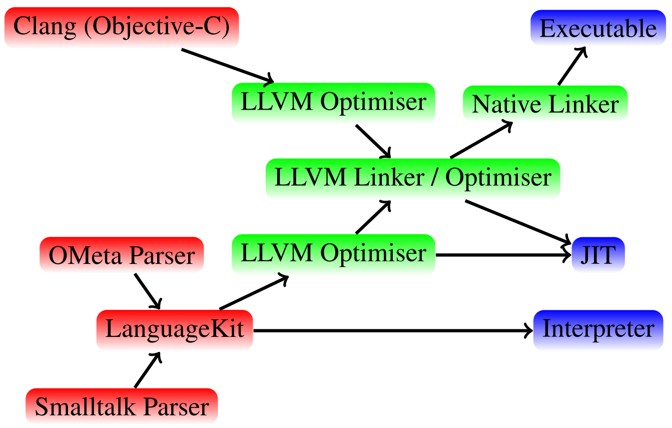


Figure 1. LanguageKit Compiler Architecture.

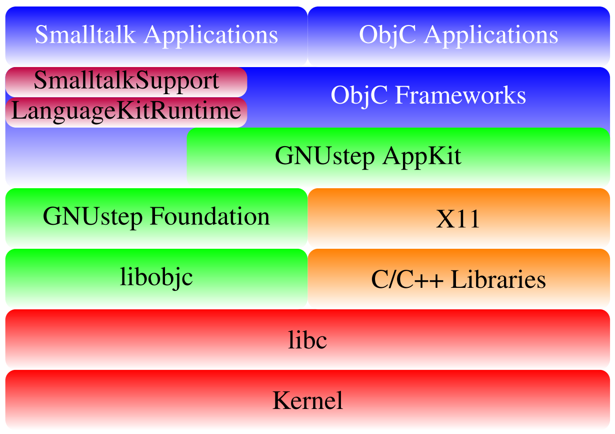


Figure 2. LanguageKit execution architecture.

level static single-assignment intermediate representation. We augment the standard set of optimisations with a number of additional ones geared to both Smalltalk and Objective-C.

The LanguageKit framework and Smalltalk front end are available in the Étoilé project's subversion repository under a BSD-style license:

`http://svn.gna.org/svn/etoile/trunk/Etoile`.

LanguageKit provides three ways of running Smalltalk code:

- A simple AST interpreter.
- A just-in-time (JIT) compiler.
- A static compiler.

How these fit together is shown in Figure 1. When compiling, LanguageKit generates LLVM's intermediate representation (IR), and links in the set of small integer support functions, written in C and compiled to LLVM IR with clang, the [Objective-]C[++] front end for LLVM. The LLVM inliner can then insert these at the point where they are used, allowing fast arithmetic and making it easy to add new small integer methods that provide similar performance without modifying the compiler. We then run a set of LLVM optimisations, which transform the IR. This can then be linked to other LLVM IR from C, Smalltalk, or any other language with an LLVM front end, and then further optimised. The optimisations run on Smalltalk and Objective-C code are largely the same currently, but in the future these may diverge as we add more Smalltalk-specific optimisations.

This provides some interesting benefits. For example, it is possible to inline C functions in Smalltalk methods and potentially vice versa. Once this optimisation has taken place, the resulting IR is translated into machine code, either in-process for immediate execution in JIT mode or on disk as object code that can then be linked with other libraries.

The JIT compiler is augmented by a background static compiler that we refer to as the 'just too late' (JTL) compiler. This runs at a low priority and generates a shared library from the code that has been JIT compiled. The next time the program is run, if there have not been any changes made to the code or any dependent libraries, the shared library is run in preference to the JIT'd version. This improves startup times and reduces memory usage: the JIT compiler itself takes around 20MB of RAM, and is lazily loaded by LanguageKit if it is required, but not if simply loading statically compiled code or interpreting.

IDE support is currently very immature, although most of the features required to build a full-featured Smalltalk IDE do exist. The AST interpreter (implemented as a generic visitor on the AST) is intended to be used mainly for debugging and development. It makes it relatively easy to modify running code: simply change the AST as it is being interpreted. Bits that have not changed for a little while can then be JIT compiled.

The interpreter uses the same underlying object model as the compiler, it simply installs a stub method for each one declared on an interpreted class, which calls out to the interpreter. This means that it is possible to mix statically compiled, JIT compiled, and interpreted code, all in the same running program. For example, we can ship well-tested libraries as statically compiled binaries, run code that is currently in development in the interpreter, and replace it with a JIT-compiled version as we gain confidence in its correctness. We can always replace a method with a different implementation, so compilation does not prevent future modification.

Unlike more traditional Smalltalk environments, we use a more conventional program and process model rather than an image. For persistence, the CoreObject framework, also developed as part of the Étoilé project, provides a simple mechanism for automatic (versioned) storage of arbitrary objects.

LanguageKit inherits a design philosophy from Objective-C, which provides a Smalltalk-like environment without the need for a virtual machine by splitting the implementation into two components. The compiler interprets Smalltalk-like message sends and a few other syntactic constructs and replace them with a call to the *Objective-C runtime library* (libobjc). This library defines the binary representation of classes and other components that have no direct equivalent in C and is responsible for handling message sending.

LanguageKit was developed along with the GNUstep Objective-C runtime, which was based on our older research prototype[11, 12]. This is intended to provide both a high-performance implementation of the Objective-C object model and provide the support required for other languages. We have made some extensions to the Objective-C runtime that are potentially useful for Objective-C and also provide two extra support libraries. The LanguageKitRuntime library provides supporting functions for things like small integer arithmetic, non-local returns, and other features that may be shared between different languages using this LanguageKit. The SmalltalkSupport framework provides some extra support functions, classes, and methods that are specific to Smalltalk. These two frameworks are linked with any Smalltalk code that is compiled with LanguageKit. Other front ends may provide additional support functions.

Figure 2 shows how these fit together at run time. Note the significant amounts of non-Smalltalk code in a typical Smalltalk application. We are reusing existing kernels and windowing systems, C and C++ libraries, Objective-C frameworks, and so on, yet still allowing Smalltalk code to exist as a first-class citizen of the resulting environment.

4. Example 1: Using a C Library

Before delving into the details of the implementation, here is a small example to help show the capabilities of the system. This is a single method from a test program.

```
run [
  | queue count |
  queue :=
    C dispatch_get_global_queue: {0.
       0}.
  ETTranscript show: 'main thread' ;
     cr .
  count := 0.
  1 to: 10 do: [
     C dispatch_async: { queue .
        [
           ETTranscript show:
              'Another thread' ; cr .
           count := count + 1
        ] } ] .
  C sleep: 1.
  ETTranscript show: 'Threads used: ';
               show: count ; cr.
]
```

In this example, we use the libdispatch library, running on FreeBSD. This library implements an N:M threading model multiplexed on top of POSIX threads and providing a work-queue based model. This highlights one of the typical limitations of Smalltalk implementations that we have addressed: the difficulty of efficiently interacting with the host system's multithreading abilities.

This example is fairly trivial, but it demonstrates the concept clearly. The first line calls a C function that returns a pointer to a C opaque type. This is boxed and stored in a local variable. It is then used as an argument to a second C call, `dispatch_async()`, which takes a `dispatch_queue_t` and a block as arguments.

This form of FFI, calling C code, was not part of our original design. We intended to use Objective-C[++] as our foreign interface layer in all cases. After writing the code to send Objective-C methods, calling C functions was a trivial extension and so we incorporated it.

The `ETTranscript` class is implemented in Objective-C, wrapping the standard C I/O capabilities. The compiler does not need to know that this is an Objective-C class. It determines the class and method information by using runtime introspection and so does not need access to headers when interacting with classes (or objects) implemented in Objective-C.

Although it is possible to call C directly, as this example shows, we more commonly wrap C libraries in Objective-C before using them in Smalltalk. This allows the Smalltalk code to only deal with high-level abstractions, rather than leaking low-level C abstractions up into a high-level language. In other approaches to FFI, this kind of abstraction is possible, but results in individual Smalltalk methods containing multiple round trips in and our of the VM. We do unboxing when assigning a value to an instance variable that was declared in Objective-C with a non-object type. This means that we can reduce the amount of boxing and unboxing required with well-designed interfaces.

Note that this example contains a race condition: it does not update the `count` variable atomically. This is intentional: running this example a few times—especially on a multi-core system—will demonstrate that it is exhibiting real parallelism, because some of the updates will be lost.

5. The Object Model and Reflection

Every Smalltalk class compiled by LanguageKit is indistinguishable in the resulting binary—in memory or in object code—from a class written in Objective-C. Classes in this implementation are represented by a C structure: `struct objc_class`. As in Smalltalk, classes are also objects. Every object or class begins with a pointer to one of these structures, referring to the object's class (or the class' metaclass).

The class structure contains all of the metadata related to the class, including the types and offsets of all instance variables and methods. Note that in this world not everything is an object: Objective-C code may declare variables, including instance variables, to be any C type. This is usually transparent to Smalltalk code, which gets these values boxed and unboxed automatically as required, but can be accessed via introspection information for cases where it is important. We also expose some functionality that is not available in Objective-C. For example, Objective-C does not contain the notion of class variables, even though these are exposed in the runtime.

We do not aim to implement the Smalltalk-80 libraries and so provide different reflection APIs to a traditional Smalltalk, however it is possible to implement things like method dictionaries as thin wrappers around our reflection support if these are desired. The EtoileFoundation framework[2] implements mirrors[9] on top of our reflection APIs, demonstrating their flexibility.

Reflection is one of the most important aspects of Smalltalk. Our implementation supports reflection down to the method level, but does not support submethod introspection or reflection for compiled code. If the AST is available, either loaded from a serialised form or from the source code, then it is possible to inspect and modify the code for a method, recompile and then attach the new method.

It is not, however, possible to modify the code in existing stack frames, unless they are running interpreted code. This is a difficult problem in general for compiled languages, because optimisations can make it very difficult to map state from one version to another. For example, local variables may be stored in registers or have their values recomputed at different points in the generated code. In future versions, we aim to support this in a subset of possible cases, to make development easier.

Objective-C introspection and reflection is accomplished via a set of functions that call into the runtime library. We have wrapped several of these in a set of classes for easier access from Smalltalk. It is possible, for example, to replace (at run time) a method in an existing class with a block. This involves a small trampoline that is written in assembly and copied by the runtime, which transforms a call frame created for calling a method into one expected by a block.

The following snippet is valid in this dialect of Smalltalk and demonstrates a trivial method being added to the class at run time.

```
addMethod [ count |
  self class
     addInstanceMethod: #testMethod
             fromBlock: [ :self |
            count := count + 1.
            ETTranscript show: count;
                         show: self;
                            cr
         ].
  self testMethod.
]
```

[2] http://svn.gna.org/viewcvs/etoile/trunk/Etoile/Frameworks/EtoileFoundation/

Executing this code fragment involves running code written in four different languages. The snippet itself is written in Smalltalk. The `+addInstanceMethod:fromBlock:` method is implemented in Objective-C in the EtoileFoundation framework, as a category on the `NSObject` class. It is a short method, which calls three runtime library functions written in C.

The first of these, `imp_implementationWithBlock()` copies a fragment of hand-written assembly (3-7 instructions, depending on the architecture) and a pointer to the block structure and the function that invokes it into a new allocation. The returned value is a small trampoline function. When invoked, it rearranges the arguments slightly and calls the block function.

This is required because methods and blocks both take a hidden first argument, the receiver. Methods also take a hidden second argument, the selector. The trampoline moves the receiver to where the block expects its first explicit argument to be (typically in a different register) and then replaces the first argument with the block pointer. It does not alter the call frame in any other way, and so is very fast.

The remaining two functions determine the correct type encoding for the method based on the block and attach it to the class. For completeness, the implementation of `+ addInstanceMethod:fromBlock:` is shown below.

```
+ (BOOL)addInstanceMethod: (SEL)
    aSelector fromBlock: (id)aBlock
{
     IMP imp =
        imp_implementationWithBlock(
        aBlock);
     if (0 == imp) { return NO; }
     char *encoding =
        block_copyIMPTypeEncoding_np(
        aBlock);
     class_replaceMethod(self,
        aSelector, imp, encoding);
     free(encoding);
     return YES;
}
```

It is also possible to add methods to a single object. This is accomplished by the runtime inserting a hidden class just for that object. This class will not be returned in response to a `-class` message sent to the receiver and is regarded as an implementation detail. A similar mechanism allows new properties to be attached to existing instances. This is the same technique employed by the Self VM[24] and the V8 JavaScript VM[4].

6. Message Sending

There are two traditional mechanisms by which message sending in Objective-C is implemented, and a third supported by the GNUstep Objective-C runtime used by this implementation. The original implementation, from the StepStone and NeXT implementations, involves calling an `objc_msgSend()` function, which takes the receiver and selector as the first two arguments and calls the corresponding method.

The GCC runtime did not adopt this mechanism, because it is impossible to implement in portable C: it requires a trampoline function that preserves the state of the call frame when invoking the method. Instead, the GCC approach splits message sending into two parts. The first part calls `objc_msg_lookup()`, a function that maps a receiver and a selector to a function pointer referring to the method. The second part then calls the method.

The GNUstep runtime supports both, as well as an extended version of `objc_msg_lookup()` that returns a cacheable structure. This is typically used in loops, where avoiding the repeated lookup when sending the same message to multiple objects can be advantageous. We also cache all message lookups where the receiver is `super`, as the result of these lookups changes infrequently. The method pointer returned by `objc_msg_lookup()` can not be safely cached because it is just a function pointer: if a new category is loaded or the method is replaced using reflection then the cache would become invalid, without there being any mechanism for clearing it.

The caching is inserted as an LLVM optimisation pass. We currently use some heuristics to determine the best sites to cache, although in future we intend to connect this to profiling information, so that we can only cache lookups on hot code paths where subsequent lookups often return the same value. The cost of a lookup is relatively small in microbenchmarks, but it varies depending on the accuracy of branch prediction. Determining when caching lookups makes sense is an open research problem. In our testing, we have discovered that polymorphic inline caching[15] is less useful on modern hardware, because the cost of testing two cache entries is greater than the cost of performing the lookup again.

We can also use the cacheable structure in conjunction with speculative inlining. For small methods, where the cost of the call is significant, static languages benefit from inlining, where the body of the method is inserted into the call site. This is not possible in the general case in Smalltalk (or Objective-C) because the user is free to replace methods at run time or substitute subclasses at arbitrary points. It is, however, possible to inline methods, but wrap the inlined version in a test checking that the method that we are expecting to call is the method that we are actually calling. With the lookup cache at the call site, we can also eliminate the need for the call to perform the lookup. In contrived microbenchmarks, doing this in a loop is approximately 30% faster than calling a C function. Again, the best time to perform this optimisation is an open research problem, because the extra branch imposes a run-time overhead if it is incorrectly predicted and the inline version consumes more instruction cache space even when it is not. As the gap between memory and CPU speed has grown over the past decade, the cost of a cache miss has grown to hundreds of clock cycles, so a few extra cache misses in a procedure can impose a performance penalty far greater than the gain from all of the optimisations performed by the compiler on the same piece of code.

Most implementations of Smalltalk provide some special handling for small objects: those embedded inside a pointer. In our 32-bit implementation, we provide support for one such class, used for storing 31-bit signed integers. On 64-bit platforms, we provide seven. Currently, four of these are used. We support 61-bit signed integers and two kinds of double-precision floating point values[3]. We also support strings of up to seven ASCII characters. These frequently appear in file paths, JSON and property list dictionary keys, and several other places: GNUstep creates 20 of these just loading the library, before launching the application when constructing file paths for configuration and library locations.

A typical XML, JSON, or property list parser generates far more. These also provide a convenient speedup in dictionary lookups. Dictionaries, implemented in Objective-C, perform pointer comparison on their keys before sending an `-isEqual:` or `-compare:` message. Because two identical short strings will have the same pointer value, we get to check them for equality in a single instruction, rather than requiring a message send, and we save some cache.

Operations on integers should ideally be fast. Other Smalltalk implementations perform a number of tricks to ensure that this is the case. In a naïve implementation, where every integer add or multiply required a message send, we would have very slow code. To avoid this, we move the small object check into the caller and then call an inline function, written in C, that does the arithmetic and overflow check. The LLVM optimiser will inline these calls before code generation.

One obvious case of impedance mismatch occurs in method types. In Smalltalk, the arguments to and result from any method are objects. In Objective-C, this is not the case. Worse, the types are not unique to a given selector. It is possible to have multiple methods with the same name but with different parameter types. Because we are setting up traditional call frames for method invocations, this can result in stack corruption if we get it wrong.

This is actually a problem in Objective-C, where calling a method with the wrong signature is undefined behaviour, yet it is impossible for a compiler to statically check whether it will happen, in the general case. Other languages that interoperate with Objective-C solve this in the same way

[3] One where the last bit is repeated, one where the last two are repeated. Thanks to the developers of Smalltalk/X and Cincom Smalltalk for proposing these.

as Objective-C; by requiring explicit type annotations for disambiguation.

To eliminate this problem, we modified the behaviour of the GNUstep runtime so that selectors are only treated as equal if they have both the same name and type encoding. Selectors are interned on load and method lookup is based on the unique number assigned when this occurs, so this adds no cost to method lookup. This means that an attempt to call a method with the wrong types would have the same effect as calling a nonexistent method. We also provide a callback in the runtime for the case where a method does exist but with the wrong types. This can be used to print an error message or throw an exception on type mismatch.

The NeXT / Apple runtime does not store type information with selectors at all. The GCC runtime does, but does not use this information for dispatch. This means that there are several cases where either the GCC or Apple runtime will silently corrupt the stack, whereas the GNUstep runtime will throw an exception.

We take this a step further for Smalltalk. If we have determined that a selector has multiple type encodings registered then we use the extended lookup function mentioned earlier, with an untyped selector. This returns a cacheable structure containing, among other things, the type encoding of the real method[4]. We can then branch to the part of the code that generates the correct call frame.

Note that this is a simpler case than running in a JVM as in [8] because our underlying object model was not intended to allow a single class to implement two methods with the same name but different types. It does now as an implementation detail, but this is not exported in any languages that target this model and is undefined behaviour if it happens accidentally. This is less of a requirement for late-bound dynamic languages than for ones with more rigid type systems, such as C++ or Java, because it is relatively easy to have multiple code paths within a method depending on the types of the objects.

7. Memory Management

C is infamous for its manual memory management, while Smalltalk is well known for supporting accurate garbage collection. Reconciling these two models is a nontrivial problem, especially when we wish to be able to pass pointers to Smalltalk objects into C code.

Our current implementation supports two models: conservative garbage collection and automatic reference counting. The former requires all Objective-C code to be compiled in garbage-collected mode, while the latter interoperates with Objective-C code using either manual or automatic reference counting.

It's important when discussing garbage collection not to conflate garbage collection (a goal) with tracing (a technique for implementing this goal). Automatic garbage collection means that objects are automatically deallocated when they are no longer required. There are a number of techniques on a spectrum for implementing this goal, with automatic reference counting plus cycle detection at one extreme and tracing collectors at the other end. [7] informs us that these approaches are both expressions of the same general algorithm.

Approaches at the reference counting end have a number of advantages, including deterministic performance and, with the low cost of atomic operations on modern systems[5] a relatively small cost in the acyclic case.

Most of our use involves the second mode: the semantics of Objective-C garbage collection are poorly defined and easy to break. For example, storing an object pointer in some memory allocated with `malloc()` or in a `static` variable declared in a C compilation unit will cause it to escape from the garbage collector's visibility and be prematurely deallocated, leaving dangling pointers, but often won't give any compiler warnings.

In automatic reference counting mode, the Objective-C memory model is improved, with explicit casts required to take pointers out of the managed world. In practice, we find that reference counting with weak pointers for explicit cycle detection are mostly sufficient. We are able to elide reference count manipulations for most on-stack assignments, making reference counting very cheap, in the absence of cycles. Given the large quantity of shipping Objective-C code on Mac OS X and iOS that manages memory efficiently with *explicit* (not automatic) reference counting, we observe that manual cycle breaking is possible for most programmers, although certainly not an ideal situation.

We have implemented an experimental cycle detector based on the concurrent cycle detector proposed by [6], but have not yet enabled it. This will likely appear in a future version of LanguageKit as an option. This works by adding objects that have their reference count decremented without being deallocated to a temporary set. If they are not deallocated within a certain amount of time then the cycle detector attempts to see if every reference to them can be accounted for by circular references and deleting the object if this is the case.

The disadvantage of the implicit cycle detector is the same as that of a tracing garbage collector: it makes object deallocation less deterministic. With reference counting and explicit cycle breaking with zeroing weak references, it is relatively easy to write code that does not leak memory, yet still deallocates objects at deterministic points in time.

The memory model used with automatic reference counting in Objective-C provides a lot of potential for future improvements. Conceptually, it makes object pointers distinct to C pointers and requires an explicit bridging cast to move

[4] If the class implements two methods with the same name but different types, the one that is returned by this lookup is undefined.

[5] An atomic increment on an Intel Core i7, in the uncontended case, costs three times as much as a non-atomic increment

between the two worlds. This makes it—theoretically, at least—possible to implement accurate garbage collection for objects that have not escaped into the C world of untyped memory.

8. Blocks

The current version of LanguageKit implements blocks using the same ABI as Apple's block extension to C. This allows blocks that come from Smalltalk or [Objective-]C to be used interchangeably. Our first versions predated blocks in C, and so used a custom BlockClosure class, responding to a similar set of messages to the Smalltalk-80 version. We have since replaced this with a set of methods attached to the `NSBlock` class, which is the superclass of the block classes in our implementation.

This allows messages such as `-whileTrue:` to be sent to blocks from either Smalltalk or Objective-C. As an optimisation, it is possible to perform an AST transform that turns `-whileTrue:` messages into loops, allowing both the tested block and the body to be inlined into the caller. This optimisation is not run by default, because it alters the semantics of the program and should only be used in code where performance matters more than flexibility. These cases are quite rare, as our interoperability model makes it easy to sink to a lower level of abstraction and use C or even assembly for these cases.

Blocks, like classes, are represented as a structure with a fixed layout and a class pointer as their first field. One of the fields is a function pointer for the function called when the block is invoked. When a block is invoked from Objective-C or C code, this function is called directly. From Smalltalk code, it is invoked via the `-value:` family of methods. This allows Smalltalk code to use any object that implements this method as if it were a block of the correct type.

Blocks are initially allocated on the stack. In the vast majority of cases, blocks do not persist beyond the lifetime of their enclosing scope. When they do, they are copied to the heap. Accesses to all bound variables happen via an indirection layer in the block structure itself. When the block is promoted to the heap, all bound variables are also promoted, in reference-counted structures. The reference counting is required because a variable may be referenced by multiple blocks. This indirection layer theoretically provides a performance penalty, but not one that we have been able to measure. It does, however, make certain categories of optimisation harder by complicating alias analysis.

The cost of copying a block to the heap, which happens whenever the block is assigned to a variable that is not on the stack, is relatively high, but this is offset by the fact that it is also relatively rare.

Currently, the back end in LanguageKit supports blocks with arbitrary argument types, but the Smalltalk front end does not. In a future version, we will implement explicit type annotations for instance variables as well as block and method arguments, to better facilitate interoperability. An example where this is needed is the iteration methods on many Objective-C collection classes, which expect a block that takes a pointer to a boolean value as an argument, allowing the block to stop iteration. A correct implementation for interoperability would provide a mechanism for writing to this value. Simpler examples include blocks taking primitive C types as arguments.

One issue in Smalltalk makes blocks somewhat difficult to implement: the fact that they are allowed to return from the scope in which they were declared. We implement this using the same 'zero-cost' model as Objective-C exceptions. Something similar is essential because a block may be passed through stack frames containing C or C++ code before it returns and so must allow all of these to run cleanup code. This is a difficulty implicit in the language, not just in our implementation: however you implement non-local returns, the VM, runtime, or compiler must include some way of unwinding the call stack. It is more complex in our case because the stack may contain frames from other languages.

Unfortunately, the 'zero-cost' comes from the fact that it costs nothing when you don't use it. Unwinding the stack in this way is actually quite expensive. We can avoid this cost in the most common cases if we inline the block, but unfortunately it is unavoidable in the general case without losing the ability to place code from other languages on the stack in between our Smalltalk stack frames.

That said, there are some special cases where performance could be improved. For example, the Objective-C collection iteration block types described earlier take a pointer to a boolean value for aborting the iteration as the final argument. It would be possible to implement the non-local return in these cases by relying on the cooperation of the intervening method respecting the behaviour of this last parameter. It is not clear whether it is worth optimising for special cases such as this.

9. Outside the Box

We regard Objective-C, and Objective-C++, as our default foreign function interface. These languages allow the creation of classes with methods that call C and C++ functions directly. We also make use of this capability to use SourceCodeKit, which uses libclang (a C wrapper around a C++ library) to parse and inspect the contents of C header files.

In Pragmatic Smalltalk, you can send a message to a fake class called `C` and have it interpreted as a call to a function of the same name as the message (with colons removed). Note that because there is no VM, there is no extra overhead for calling C functions in this way. The compiler sets up the call frame in exactly the same way that a C compiler would. In fact, it is often cheaper to call a C function from Smalltalk than it is to send a Smalltalk message.

The exception happens when boxing or unboxing is required. C types are boxed in the `NSValue` class or one of its subclasses, an OpenStep class designed for encapsulating arbitrary C types. The most notable subclass of this is NSNumber. We insert our small integer and floating point classes into the existing hierarchy below this, so that boxed integers do not require allocating on the heap, unless they overflow the 31 or 61 bits allocated within pointers.

10. Performance

The current release of LanguageKit focusses on correctness over performance and so incurs some performance regressions over previous versions, which we aim to address in future versions. In spite of this, Pragmatic Smalltalk compares reasonably with our baseline: GCC 4.2.1 compiling Objective-C code. This version is somewhat old, but was widely used for shipping code and is the last version under the GPLv2 and so the last version shipped with several operating systems. Brief testing indicates that newer releases do little to improve Objective-C performance and make several regressions. Objective-C code compiled with clang benefits from a number of our enhancements, so would not make a sensible baseline, as it is a moving target.

Most of the performance improvements that we have made in Smalltalk are also of use in Objective-C when compiling with clang. We have reduced the cost of message sending to somewhere between 1.5 and 2 times the cost of a C function call, depending on the architecture and mechanism used. In older versions of the runtime this cost was 3-4 times that of a C call. This mechanism is used by both Objective-C and Smalltalk and so both Objective-C and Smalltalk benefit from this speedup.

Our implementation currently performs very poorly in any cases involving floating point arithmetic, as we have not optimised this at all. Every floating point operation involves a message send (while small integer operations are inlined) and so is typically at least a factor of ten slower than the equivalent code in C.

Small integer arithmetic is comparable in performance to C integer arithmetic. A recursive fibonacci benchmark indicates that we achieve something around 20% of C performance, however profiling indicates that most of this is due to the requirement of checking whether we need to increment the reference count of the returned object. It's also worth noting that, in this benchmark, the Smalltalk version becomes significantly slower for larger numbers while the C version does not: it much more quickly gives the wrong answer as the result of undetected integer overflow.

11. Example 2: Scripting with Smalltalk

Étoilé includes a very small framework called ScriptKit, which exposes objects from applications via the distributed objects mechanism for external scripting. It also include an AppleScript-inspired class called Tell, that implements one method: +application:to:, which is used as shown in the following example. The method requests the dictionary of scriptable objects from the application and passes it to the block.

```
Tell application: 'Typewriter'
  to: [ :dict | app win |
        app := dict objectForKey:
          'Application'.
        win := app mainWindow.
        app sendAction: #performClose
                    to: win
                  From: nil
      ]
```

This example will close the main window of the Typewriter application, the rich text editor that forms part of Étoilé. LanguageKit is also used for scripting in the 3SUM[1] modelling application. This goes one step further and parses a set of command-line arguments as a message expression, allowing users to write Smalltalk code inside shell scripts, with the program itself appearing to be an object.

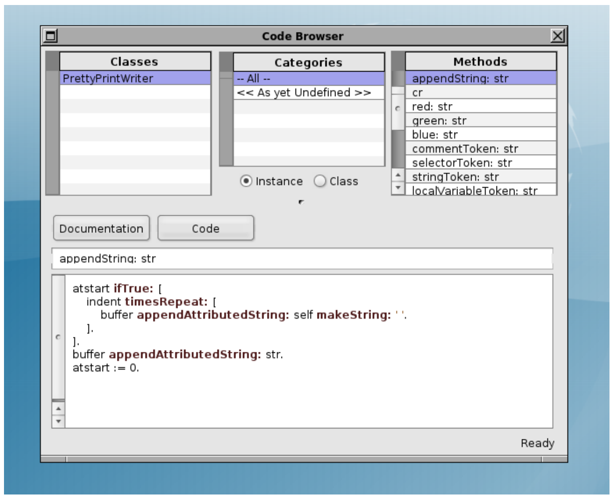
In a lot of applications, the line between scripting code and real application code is blurred. The Firefox web browser, for example, is written in a mixture of JavaScript and C++. JavaScript is used for scripting and plugins, but also for significant parts of the user interface. LanguageKit aims to encourage this development model by eliminating the overhead from transitions between languages. This allows applications to be rapidly prototyped in Smalltalk and performance-critical parts to later be rewritten in C if they are determined to be too slow.

In particular, it is possible using LanguageKit for code to begin life as interpreted or JIT-compiled 'scripting' code inside an application, but then be statically compiled and shipped with a future version. End users will be totally unaware of the presence of Smalltalk code: even if they open up a class browser in the application's scripting environment, they will just see classes with methods and be unaware of whether they were implemented in Objective-C, Smalltalk, or some domain-specific language.

12. Example 3: The CodeMonkey IDE

The CodeMonkey IDE is a work-in-progress integrated development environment that is intended to be used for Smalltalk and Objective-C development. It is written in a mixture of Smalltalk and Objective-C. For example, the code required for syntax highlighting Smalltalk is written in Smalltalk, while the code required for syntax highlighting Objective-C uses the clang Objective-C parser.

CodeMonkey development is currently slow because it has not been an active focus for developers (it has received a total of approximately two person-weeks of developer time), but it demonstrates the use of Smalltalk in a program using many C, C++ and Objective-C libraries.



```
appendString: str
```

```
atstart ifTrue: [
   indent timesRepeat: [
      buffer appendAttributedString: self makeString: ' '.
   ].
].
buffer appendAttributedString: str.
atstart := 0.
```

Figure 3. The CodeMonkey IDE

13. Future Work

Our current implementation is usable and suffices to prove that the concept is viable, but there is significant potential for future work. An obvious area for improvement is performance. For example, we currently test whether small integer operations overflowed twice, once when determining whether to box them and then again determining whether they need explicit reference count manipulation. We do not run any of the feedback-driven optimisations by default (e.g. speculative inlining, which can provide a significant performance benefit), even though doing so would be relatively easy if we added the instrumentation when doing the first JIT compilation and then used the results in the JTL. Even with our current performance, however, this implementation is fast enough for most new code, and makes it easy to fall back to C for the few cases where it is not.

Another focus of future work is adding more languages to the system. We have an experimental dialect of JavaScript compiled using LanguageKit and work is underway on a dialect of OMeta[25]. Once the latter is completed, it will become a lot easier to add new front ends for experimental languages.

Smalltalk is not expected to be the end result of our work. In the future, programming languages are going to need to deal with increased levels of concurrency and with heterogeneous multiprocessor systems. We intend to explore various extensions to our system to allow languages to develop apace with hardware, without requiring any legacy code to be rewritten faster than its natural replacement rate.

14. Related Work

Our C callout mechanism is heavily inspired by the notion of Aliens from Newspeak[10].

Two other notable attempts have been made to couple Objective-C with a higher-level language. F-Script[2] and MacRuby[3] both target Apple's implementation of Objective-C, which limits their flexibility as they are unable to extend the Objective-C.

F-Script is similar to the earlier StepTalk scripting interface for Objective-C, which began life on GNUstep and was ported to OS X. Both provide a scripting environment where a dialect of Smalltalk can be used to send messages to Objective-C objects, allowing introspection information to be presented to the user. The existence of these tools provided motivation for our development, as they are commonly used not just for scripting but also for prototyping. We aim to make it possible to use Smalltalk in production where StepTalk or F-Script would only have been used in the prototyping phase.

MacRuby provides some integration between Ruby and Objective-C, for example using Objective-C classes to represent things like strings, just as we do. MacRuby began after our work, but implements some similar ideas. They only support the garbage collected mode for Objective-C, which is no longer encouraged for new development on OS X or GNUstep and was never supported for iOS. The MacRuby approach retains a Ruby class model, so certain operations need conversions. In contrast, we compile Smalltalk and Objective-C to binary code with the same underlying representations for all common features.

RubyCocoa uses the BridgeSupport framework, which statically provides type information for things like C functions, rather than dynamically extracting it from C headers. The RubyCocoa project is also limited by the requirement to work on the Apple Objective-C runtime, which contains a number of limitations. For example, it does not allow safe caching of method lookups, nor does it provide a cheap mechanism for checking the types of a method at run time.

Other work has been done on improving Smalltalk's ability to interoperate with other languages. Most notably, the NativeBoost[21] work has made it possible to attach C functions as methods to Smalltalk objects. This works by creating a small assembler in Smalltalk—also used for JIT compilation—using it to create method stubs that perform unboxing and create C call frames.

NativeBoost attempts to keep Smalltalk in its isolated world, but make it easier to punch holes in the edges. This has several issues from a programmer perspective. Most notably, retaining the image abstraction but having some things outside the image (and therefore not persistent) is problematic, especially when they can be encapsulated in normal Smalltalk classes. There are also issues with garbage collection, which are the mirror image of the ones that we encounter: the Pharo garbage collector must be told not to relocate objects while C code may hold references to them.

While promising, NativeBoost falls into the typical Smalltalk trap of avoiding using existing resources. By writing assembly or machine code natively, it requires the entire optimisation stack to be written in Smalltalk. This is a far from trivial amount of code: The standard set of LLVM optimisations include over 80,000 lines of C++ code, excluding complex supporting logic and the register allocator. The target-specific logic in LLVM weighs in at around 160,000 lines of C++. Reimplementing all of this in Smalltalk would be a considerable amount of work. Simply extending it with some Smalltalk-specific optimisations is much simpler.

Smalltalk/X [14] provides a potentially more interesting approach, performing a source-to-source translation of Smalltalk into C and then compiling this with the system's native C compiler and loading the resulting shared library. This has some significant disadvantages, for example the inability to insert Smalltalk-specific optimisations into the C compiler and the requirement to round-trip via the filesystem for JIT compilation. This approach does make interoperability easier, as Smalltalk methods can contain inline C that is passed through to the C compiler, something that our implementation can not do.

Three other notable projects have aimed at providing interoperability with other environments. Redline[5] and IronSmalltalk[23] compile Smalltalk to the Java and .NET bytecode, respectively. These both target other virtual machines, both of which were—directly or indirectly—inspired by the Smalltalk VM. Both have had one significant advantage over Smalltalk in terms of uptake: large companies devoting huge marketing and development budgets to producing them.

The third project is Amber Smalltalk (formerly JTalk[20]), which translates Smalltalk into JavaScript that runs on the V8 JavaScript virtual machine. JavaScript is very similar to Self in terms of object model, which makes it a natural target for Smalltalk. Amber provides similar advantages to our approach: interoperability with widely deployed environments, allowing gradual deployment of Smalltalk code. The only difference is the target, with Amber aiming to run code in web browsers.

15. Conclusion

We have shown that it is possible to retain most of the flexibility of a high-level language like Smalltalk, as well as achieving good performance, while managing to achieve a high degree of interoperability with low-level languages. We believe that there is a potential for significant performance improvements, on the order of a factor of two to four, without too much effort.

By making it easy to mix C and Smalltalk, we make it easy to choose the correct tool for the job, writing high-level code in a high-level language and performance-critical code in a low-level language. We have shown that incremental migration to Smalltalk from low-level languages is possible by providing a framework for easy interoperability between new Smalltalk code and legacy C code.

We have also shown that it is possible to achieve reasonable performance with a late-bound dynamic object oriented language. Our current implementation lacks a number of important optimisations, in particular type inference for sequences of integer or floating point operations, which gave a significant speedup in HiPE[18]. It is also far more conservative in memory management than is strictly necessary and makes no attempt to inline reference counting operations. In spite of these limitations, we achieve performance very close to Objective-C for common operations, such as string manipulation[6] and a tolerable speed for arithmetic-heavy code.

Our current approach lacks some of the advantages of Smalltalk. The most obvious of these is debugging. Our current implementation emits very sparse DWARF debugging information and so is fairly limited in terms of debugging support even in comparison to C, and therefore a long way behind the state of the art for Smalltalk circa 1980. This is currently the focus of ongoing work. Once this is done, then implementing things like `thisContext` making use of debug metadata become possible. In our current implementation, run-time introspection is only available for objects and variables bound to blocks, not for activation records.

Closely related is the rest of the IDE. In traditional Smalltalk implementations, the IDE is closely integrated with the execution environment. GNU Smalltalk is the major exception, and provides a model close to ours. Building a good IDE and debugger is beyond the scope of the LanguageKit project, but building these tools on top of LanguageKit is a goal of Étoilé.

16. Acknowledgements

LanguageKit would not have been possible without feedback from a number of people, most notably Nicolas Roard, Quentin Mathé, Niels Grewe and Eric Wasylishen. Eric also wrote the initial version of the interpreter and Niels added support for x86-64. It would certainly be far less useful if not for Guenther Noack, who wrote the test suite and filed numerous bugs. More recently, Mathieu Suen has been working on adding support for debug information and on the OMeta front end.

Portions of this work were sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] 3sum. `http://sourceforge.net/apps/mediawiki/mus3/`. Accessed: 08/06/2012.

[2] F-script. `http://www.fscript.org/`. Accessed: 08/06/2012.

[6] These are difficult to benchmark fairly, because much of the execution time is spent in Objective-C[++] code.

[3] Macruby. `http://macruby.org/`. Accessed: 08/06/2012.

[4] V8 javascript engine design elements. `https://developers.google.com/v8/design/`. Accessed: 08/06/2012.

[5] S. Allen. Redline smalltalk. In *2011 International Smalltalk Conference*, 2011.

[6] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In J. L. Knudsen, editor, *Proceedings of the Fifteenth European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235, Budapest, Hungary, June 2001. Springer-Verlag.

[7] D. F. Bacon, P. Cheng, and V. T. Rajan. A unified theory of garbage collection. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications*, pages 50–68, Vancouver, British Columbia, Oct. 2004.

[8] A. Bergel. Reconciling method overloading and dynamically typed scripting languages. *Computer Languages, Systems & Structures*, 37:132–150, 2011. doi: 10.1016/j.cl.2011.03.002.

[9] G. Bracha. Mirrors: design principles for meta-level facilities of object-oriented programming languages. pages 331–344. ACM Press, 2004.

[10] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, and E. Miranda. The newspeak programming platform. *Design*, pages 1–15, 2008.

[11] D. Chisnall. A modern Objective-C runtime. *Journal of Object Technology*, 8(1):221–240, Jan. 2008.

[12] D. Chisnall. A new objective-c runtime. *Communications of the ACM*, Sept. 2012.

[13] B. J. Cox and A. J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[14] C. Gittinger. Guided tour through smalltalk/x. In *2011 International Smalltalk Conference*, 2011.

[15] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, London, UK, 1991. Springer-Verlag. ISBN 3-540-54262-0.

[16] D. H. H. Ingalls. The Smalltalk-76 programming system design and implementation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 9–16, New York, NY, USA, 1978. ACM. doi: http://doi.acm.org/10.1145/512760.512762.

[17] D. H. H. Ingalls. Design principles behind Smalltalk. *Byte Magazine, special issue on Smalltalk*, August 1981.

[18] E. Johansson, M. Pettersson, K. Sagonas, and T. Lindgren. The development of the hipe system: Design and experience report. *Journal of Software Tools for Technology Transfer*, 4 (4):421–436, August 2002.

[19] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[20] N. Petton. Iliad & jtalk. In *2011 International Smalltalk Conference*, 2011.

[21] I. Stasenko. Native boost. In *2011 International Smalltalk Conference*, 2012.

[22] C. Thacker, E. McCreight, B. Lampson, R. Sproull, and D. Boggs. Alto: A personal computer. Technical report, Xerox PARC, 1979.

[23] T. Todorov. Running smalltalk on top the .net dlr. In *2011 International Smalltalk Conference*, 2011.

[24] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA*, pages 227–242, 1987.

[25] A. Warth. *Experimenting With Programming Languages*. PhD thesis, UCLA, 2008.

Refactoring Support For Smalltalk Using Static Type Inference

Martin Unterholzner
Lifeware SA, Switzerland
martin_unterholzner@hotmail.com

Abstract

Refactoring is a crucial activity in agile software development. As a consequence, automated tools are expected to support refactoring, both for reducing the developer's effort as well as for avoiding errors due to manual changes. In this context, the chosen programming language has a major impact on the level of support that an automated refactoring tool can offer. One important aspect of a programming language concerning the automation of refactoring is the type system. While a static type system, present in languages such as Java, provides information about dependencies in the program, the dynamic type system of the Smalltalk programming language offers little information that can be used by automated refactoring tools.

This paper focuses on the challenges in the context of refactoring raised by the dynamic type system of Smalltalk. It highlights the problems caused by the absence of static type information and proposes the use of static code analysis for performing type inference to gather information about the dependencies in the program's source code. It explains the mechanism of the static code analysis using sample code and presents a prototype of an enhanced refactoring tool, which uses the structural information extracted through static code analysis. Empirical samples build the base for evaluating the effectiveness of the approach.

1. Introduction

Frequent requirement changes challenge the design of a software system and eventually lead to the erosion of its structure [35]. Modern agile software development methodologies, such as Extreme Programming, propose continuous restructuring of the code as a countermeasure to avoid the decay of the architecture [2, 12]. The activity of restructuring the code without altering the behaviour of the program is referred to as refactoring. In his Ph.D thesis [24], Opdyke listed a catalogue of refactorings and addressed the issue of their safety in the context of preserving program semantics. In his pioneer work, Johnson implemented a refactoring tool for the Smalltalk programming language [30]. By reducing the efforts needed for refactoring as well as the number of defects introduced during restructuring, refactoring tools ease frequent structural changes and, thus, software maintainance [5, 20, 23]. Today's IDEs (Integrated Development Environments) ship with a number of tools for refactoring [7].

IWST 2012 August 28th, 2012, Gent.
Copyright © ACM [to be supplied]. . . $10.00

The degree to which a tool can provide refactoring support depends on the programming language, because the type system plays a major role in refactoring. A static type system, present in programming languages such as Java, reveals information about the program's structure, which can be used by refactoring tools [16, 27]. In contrast, a dynamic type system, such as the one of Smalltalk, provides no precise information to refactoring tools about dependencies in the program. Therefore, in some cases, tools are unable to identify the code affected by a refactoring [38]. For instance, overloaded method names in Smalltalk cannot be distinguished by refactoring tools, which hinders the automation of various refactorings, such as the rename method refactoring. This disadvantage raises challenges regarding the maintenance of large software systems written in Smalltalk by forcing developers to carry out refactorings manually.

To avoid the need for manual refactoring, this paper proposes the enhancement of refactoring tools in Smalltalk with type information extracted through static code analysis. It illustrates the mechanisms of the static code analysis by means of a sample program. The solution approach is implemented as a prototype of selected enhanced refactoring tools for the VisualWorks Smalltalk programming environment. Empirical analyses build the base for evaluating the effectiveness of the prototype and of the approach.

The remainder of this paper is organised as follows:
Section 2 focuses on the challenges regarding refactoring automation raised by the dynamic type system in Smalltalk, exemplified through a sample program. **Section 3** describes the details of the static code analysis used to extract the type information. Sample code snippets illustrate the single steps performed during the code analysis. The section concludes with the description of the components used for building the prototype. **Section 4** assesses the effectiveness of the approach and of the prototype, based on empirical statistics. **Section 5** concludes this paper by mentioning the contributions of the presented research.

2. Problem Description

It is common for the customer's requirements to change during the entire life-cycle of a software product, which - according to Lehmann - is caused by a changing operational domain [18]. The impact of changing requirements on a software project depends on the importance given to software maintenance within the chosen development methodology. For instance, the classical waterfall model does not promote change requests after the initial requirement engineering phase. In contrast, modern agile software development processes use incremental development combined with short release cycles to gather early and continuous customer feedback. Continuous feedback allows one to quickly react to change requests. However, frequent change requests conflict with the existing software architecture by leading to "design erosion" [35] and, thus, demand for a software architecture that can be restructured and maintained easily.

For reaching a continuously evolving software architecture, Extreme Programming - one of the agile software development methodologies - incorporates refactoring as a crucial concept into the development process [2]. Refactoring is the activity of restructuring the program, without changing the behaviour that is relevant from the customer's perspective [12, 24]. Examples for widely known refactorings are: rename method, rename class, extract method, extract to component, inline method, add method parameter, and remove method parameter.

2.1 Refactoring Automation and Tool Support

There are two different contexts, in which developers do refactoring. Firstly, refactoring is done after the completion of a feature implementation, when the developer reviews the code and finds bad code smells [12]. Secondly, when the team is given a requirement for a new functionality and realises that the current architecture does not support the abstractions needed for implementing the functionality, the system has to be restructured before implementation [9, 30].

During refactoring three steps can be identified [21]:

1. finding the piece of software that needs to be restructured
2. deciding which refactoring to apply
3. performing the chosen refactoring

Automation and tool support for refactoring is crucial for turning this activity into an intrinsic part of the development process [30]. Researchers made various attempts to automate all the three steps [20, 22]. Kataoka et al. propose program invariants for finding places in the code where particular refactings are applicable [17]. Lippert and Roock as well as Ducasse et al. propose a metric-based smell detection mechanism for finding pieces in the program that are candidates for refactorings [8, 19].

In today's programming environments, however, only the last step is automated and integrated into tools, leaving the first two steps, which require decisions based on experience, to the developer. Automated refactoring tools divide their work into smaller steps, including the application of preconditions and postconditions for ensuring that the refactoring is correct and behaviour-preserving.

Regardless of the automation of many refactorings, in today's tools, in some cases the developer needs to perform also step 3 of a refactoring manually, if the tool does not provide automation of the desired refactoring.

2.2 Impact of the Programming Language

While dynamically typed languages such as Smalltalk support short release cycles by easing the quick implementation of new features (behavioural change), they bring up additional challenges regarding the application of refactorings (structural change) [9].

On the one hand, in the context of refactorings - both manual and automated - dynamic languages are less safe than statically type-checked languages, since they do not provide static type checking, which can detect type inconsistencies [9, 38]. For instance, Tip et al. use the static type system of Java to reason about the correctness of refactorings [32].

On the other hand, the absence of type information at compile time hampers finding and understanding dependencies of objects in the program [37], which raises further issues regarding the automation of refactorings, such as identifying the affected pieces of source code [9, 16, 27, 38]. These issues will be further explained by means of two concrete examples of refactorings applied to a sample program.

2.3 Sample Code

Figure 1 presents the classes of a sample program. The code listings 1, 2, and 3 show the methods of the classes. Fragments of this sample program's code are used throughout the paper for explaining the problems and solution concepts regarding refactoring. To increase legibility, the listings omit trivial code elements such as constructors, getters and setters, which are not relevant for the refactoring example, and standard objects like the `Stream` classes, which belong to the standard Smalltalk library [14].

In addition to the methods of the aforementioned classes, Listing 5 presents a code fragment, which instantiates objects. In a productive system, object instantiation is introduced, for example, through the GUI or other external interfaces such as a web service.

Listing 1. methods in the class `Movie`

```
price
  ^price

price: aNumber
  price := aNumber

name
  ^name

name: aString
  name := aString
```

Listing 2. methods in the class `Customer`

```
name
  ^name

name: aTranslation
  name := aTranslation
```

Listing 3. methods in the class `Rental`

```
movie: aMovie
  movie := aMovie

customer: aCustomer
  customer := aCustomer

numberOfDays: aNumber
  numberOfDays := aNumber

price
  ^movie price * numberOfDays

printTotalOn: aStream
  aStream nextPutAll: 'Customer name: '.
  aStream nextPutAll: customer name.
  aStream cr.
  aStream nextPutAll: 'Movie title: '.
  aStream nextPutAll: (movie name displayIn: customer
       language).
  aStream cr.
  aStream printNumber: self price.
  aStream nextPutAll: ' EUR'.
```

Listing 4. methods in the class `Translation`

```
displayIn: aString
  ^aString = 'english' ifTrue: [english] ifFalse: [german]
```

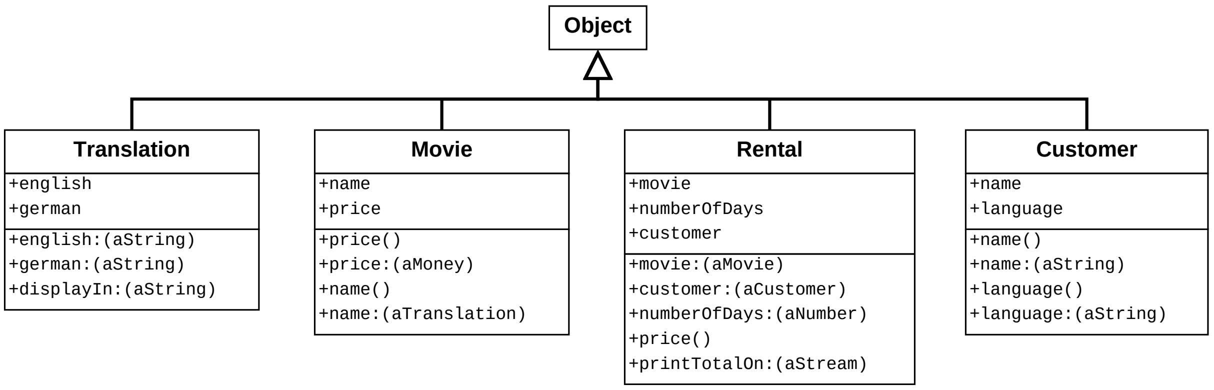


Figure 1. Original class diagram before refactoring

Listing 5. code fragment for creating instances

```
| theGodfather alice rental stream title |

stream := WriteStream on: String new.

title := Translation new.
title english: 'The Godfather'.
title german: 'Der Pate'.

theGodfather := Movie new.
theGodfather name: title.
theGodfather price: 3.

alice := Customer new.
alice name: 'Alice'.
alice language: 'english'.

rental := Rental new.
rental movie: theGodfather.
rental customer: alice.
rental numberOfDays: 2.

rental printTotalOn: stream.
stream contents.
```

The upcoming manual refactoring on this sample program emphasises the drawbacks of a dynamically typed language with respect to refactoring.

2.4 Manual Refactoring

The purpose of the first sample refactoring is to prepare the program architecture to support more than one currency in the movie rental system. It requires the following changes to the sample program to maintain semantic equivalence according to the definition of refactoring [24]:

1. creating two new classes, `Currency` and `Money`, according to Figure 2.
2. ensuring that all the writers of the variable `price` in `Movie` pass an instance of `Money` instead of a number. Consequently, the return type of its reader is changed.
3. changing all the references to the getter method `price` in `Movie` - shown in Listing 6 - according to its new return type.

Listing 6. methods in Rental after refactoring

```
price
  ^movie price amount * numberOfDays

printTotalOn: aStream
  aStream nextPutAll: 'Customer name: '.
  aStream nextPutAll: customer name.
  aStream cr.
  aStream nextPutAll: 'Movie title: '.
  aStream nextPutAll: (movie name displayIn: customer
      language).
  aStream cr.
  aStream printNumber: self price.
  aStream nextPutAll: movie price currency name.
```

As a prerequisite for this refactoring, the developer needs to know the following details:

- Which methods are affected?
- What changes does she have to make in each method?
- In which order does she have to make the changes on the affected methods? For instance, between the steps 2 and 3, the program is in an inconsistent state. Hence, the execution of both of them is an atomic operation.

While in the case of automated refactorings the tool is responsible to handle the aforementioned issues automatically, when performing manual refactoring, the programmer has to take care of them. However, the IDE is assumed to give assistance in finding the affected methods by listing all the references (`Rental>>price` and `Rental>>printTotalOn:`) to the method whose return value is about to be changed (`Movie>>price`). Giving this assistance is essential for supporting those manual refactorings in a large scale industry software project.

In Smalltalk, the refactory browser assists the programmer by listing all the methods referring to methods with the name `price`. This is accurate as long as the method name `price` has only one meaning and is not overloaded. In our code example, however, there are two different methods named `price`, of which only one is relevant for the sample refactoring. In a large scale industry environment many more method names are overloaded with numerous implementors. This is true for methods having generic names like `price, name, description` as well as names referring to domain specific concepts. For instance, in the life insurance system of Lifeware, which contains more than 37,000 classes, there exist more than 150 implementors of methods named `premium`. In such a situation, the list of references given by the refactory browser is

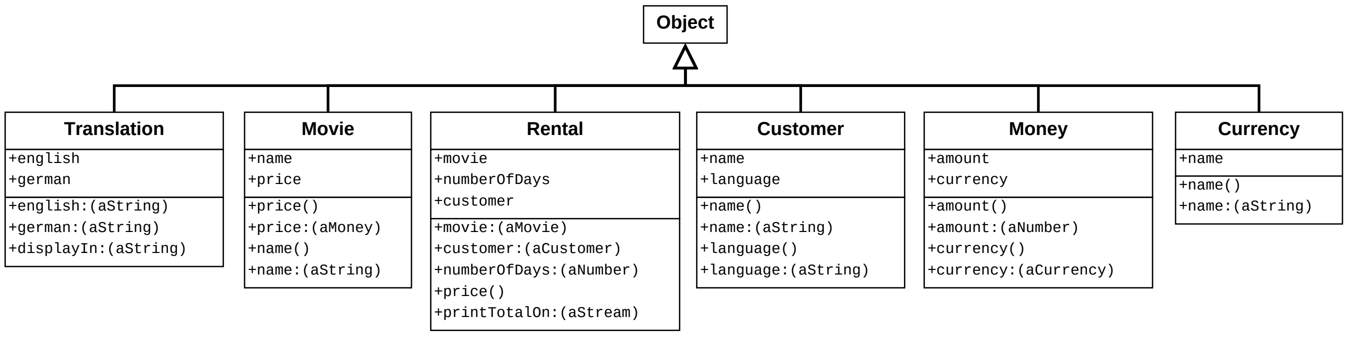


Figure 2. Class diagram after refactoring

a superset of the relevant set of references, and therefore contains a considerable amount of noise. As a consequence, the usefulness of the refactory browser for supporting such kinds of manual refactorings decreases rapidly in such a large project.

2.5 Renaming Overloaded Methods

The same problems mentioned for the sample refactoring in Section 2.4 hinder the automation of some standard refactorings, like the ones listed in Section 2.

Assume the developer wants to clarify the semantics of the method `name` in `Customer` by changing its name to `lastname`, possibly by using the automated rename method refactoring. The method `name` has two implementors - as shown in Listing 1 and Listing 2 - with different meanings, i.e., they are not used polymorphically. The automated rename, however, is unable to complete the desired change, because safely renaming a method in a dynamically typed language such as Smalltalk requires renaming all the implemented methods that have the same name, as well as changing all the senders of this method name, regardless of them not being used polymorphically. Similarly, adding or removing parameters from methods alters all the methods with the corresponding method name and all the potential references. In a dynamically typed language this is currently the only way an automated refactoring can preserve semantic equivalence due to the absence of static type checking.

3. Proposed Solution

3.1 Overview

After having studied the specific cases, where tool automation for refactoring raises challenges in dynamically typed languages, it becomes clear that improvements in this area could significantly increase the developer's productivity [20]. The desired solution is an automation level close to the one offered by Eclipse for Java, where the tool identifies not only the method names, but also the different semantics with respect to polymorphism by using the information coming from the type system.

To reach this objective, it is required to understand why the refactoring tools benefit from the existence of a type system, and which of its components are essential for identifying different semantics.

In the case of dynamically typed languages, the information available is the complete list of all the implementing classes and all the senders of one particular method name. However, the information which sender refers to which implementor is missing. For this reason, the refactoring uses a pessimistic approach for guaranteeing safety regarding behaviour preservation by assuming that each sender could refer to any of the existing implementors [30].

The following subsections present the missing concept - which sender refers to which implementor - as a graph. Using this graph, this paper shows the advantages of having access to type information extracted through static code analysis. As an example, it introduces a proof-of-concept implementation of a rename method refactoring built upon the aforementioned graph as well as on the built-in refactoring framework, which provides and manipulates the parse tree of the source code.

Figure 3 shows all the used concepts and the links between them. A grey shading of a concept means re-usage of an existing implementation, while the non-shaded components are newly developed for the proof-of-concept refactoring. An arrow connecting two concepts in the diagram indicates a dependency of its destination on its source. The upcoming subsections explain each concept in detail.

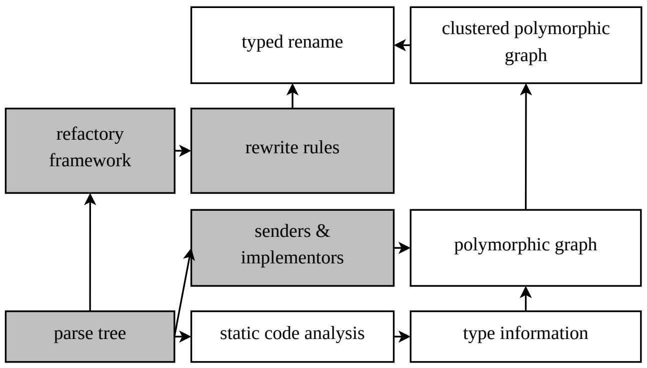


Figure 3. The implementation of the enhanced rename method refactoring requires various concepts and frameworks

In addition to the description of concepts given in this paper, the original thesis explains the development process and challenges encountered during the implementation of the prototype [33].

3.2 Missing Information

Before presenting the solution approach, this section briefly contrasts the properties of the type systems of Smalltalk and Java. Given a hierarchy shown in Figure 4 that uses the classes `Bird`, `Cat`, and `Dog`, Smalltalk allows polymorphic usage of all the classes with respect to the message `eat`, while in Java the programmer must decide at compile-time whether the receiver is either a `Mammal` or an instance of `Bird` by declaring the type of the variable.

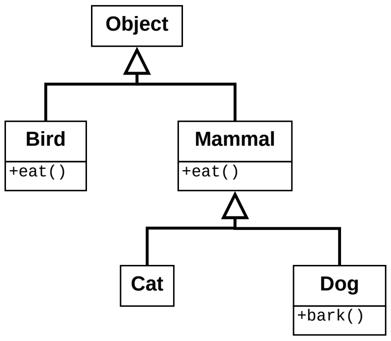


Figure 4. In Java - according to this class hierarchy - only instances of `Cat` and `Dog` can be assigned to a variable declared as `Mammal`

This difference is depicted in Figure 5 and Figure 6. They show the sets of objects allowed for polymorphic use in Smalltalk and Java as a bipartite graph. Its right hand side shows the sent message `eat`, while the left hand side shows the potential class candidates of the receiver objects: `Bird`, `Cat`, and `Dog`.

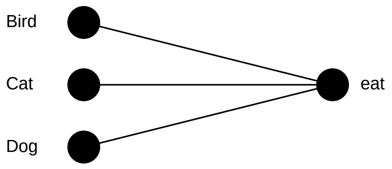


Figure 5. In Smalltalk the message `eat` can be sent to any of the objects

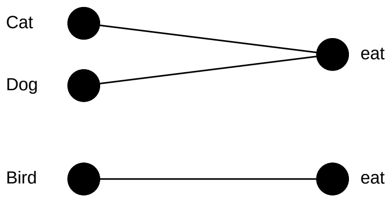


Figure 6. Java requires the programmer to decide between the two options: either the receiver is a `Mammal` or an instance of `Bird`

This bipartite graph is referred to as ”polymorphic graph” in this paper and is used to illustrate actual or potential polymorphic use. The following definition specifies its properties in more detail:

Polymorphic graph Each implementor and each sender of a method name is represented as vertex in a bipartite graph, having the implementors on one side, and the senders on the other side. The edges represent the information, which sender refers to which implementor.

This paper proposes an approach for distinguishing the set of objects for which the programming language **allows** polymorphic usage corresponding to the type constraints, depicted in Figure 5 and Figure 6, from the set of objects that are potentially polymorphically **used** in the program, which is a subset of the objects allowed by the type system of a programming language.

The remaining sections of this paper use the model of the polymorphic graph to analyse the relationship between those two sets and focus on the consequences on refactoring that arise from it.

As mentioned previously, in a dynamically typed language, the information which sender refers to which implementor is not available. Hence, for guaranteeing safety, refactoring tools consider the pessimistic case of a complete graph [30], as shown in Figure 7.

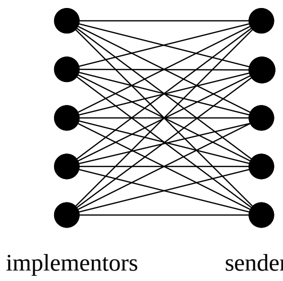


Figure 7. A complete graph indicates that any sender potentially refers to any implementor

In statically typed languages, however, the type system can provide some information about the edges of the polymorphic graph. For instance, if the receiver of the message `name` is declared as `Customer`, the type system guarantees that the receiver is not of the type `Movie`. Thus, the static type system can help by reducing the number of edges in this bipartite graph. As a consequence, potential clusters according to the following definition emerge:

Cluster Clusters are separate bipartite subgraphs, whose vertices do not share any connection with vertices belonging to any other cluster. An example is depicted in Figure 8.

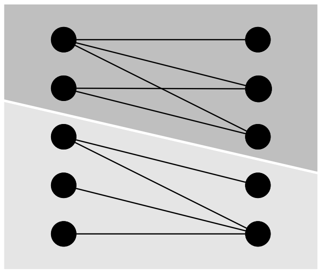


Figure 8. The type information fosters emerging clusters

If such separate subgraphs can be found, it is no longer necessary to rename all the implementors and all the senders of one method name. Semantic equivalence can safely be achieved by changing only the implementor and sender vertices contained in one cluster. This explains why the automated rename in Eclipse for Java can handle the rename of the method `name` in equivalent Java code for Listing 2.5.

In order to support advanced automated refactoring tools for dynamically typed languages, such as Smalltalk, it is required to enrich the bipartite graph with the information which edges can safely be excluded. The next subsection shows existing approaches for extracting information regarding the edges in a polymorphic graph.

3.3 Previous Solution Approaches

The following approaches aim at obtaining the type information needed to build the polymorphic graph.

3.3.1 Extension of Smalltalk with a Static Type System

Bracha and Griswold extend the Smalltalk language with a static type declaration and checking system (Strongtalk) [4]. This approach demands active support by the developer, requiring to explicitly add type declaration annotations for all the used variables.

The type checker coming with Strongtalk verifies the manually added type declarations. Using this approach, Strongtalk can benefit from the same advantages that come with statically type-checked languages such as Java. However, it also comes with the same drawback: the developer needs to do additional type declaration work, which is usually perceived as overhead. Furthermore, this approach becomes a heavy burden when migrating existing large code bases, because it is necessary to enrich the entire code base with the missing type declarations.

3.3.2 Combination of Type Declarations with Type Inference

Borning and Ingalls enhance the Smalltalk language with a type declaration mechanism combined with type inference [3]. Even though inside methods a type inference mechanism is used for temporary variables to reduce the need for type declarations, also this approach relies on active type declarations from the developer, which implies additional effort as mentioned previously. The evaluation of the implemented prototype was limited to small projects done by its developers.

3.3.3 Static Type Inference Using Pruning for Scalability

Spoon and Shivers present a demand driven type inference mechanism that uses pruning [31]. Type information is extracted only on a selected subset of program elements. To make the inference scalable, this approach trades precision of type information with scalability and performance by applying pruning that is either based on timeouts or the number of visited nodes. The evaluation - which consisted of manually checking the results of the implemented prototype - revealed good precision on a large scale project.

While having partial type information is valuable for a better understanding of the program, losing precision potentially undermines safety in the context of refactoring.

3.3.4 Dynamic Type Inference While Running Test Cases

Rapicault et al. dynamically infer the type of the message receiver by running the automated test cases available in software projects developed according to test driven development paradigms such as Extreme Programming [27]. The construction of the polymorphic graph is based on run-time information extracted from test runs. This approach avoids the pitfalls mentioned for the method before. Nevertheless, it demands a representative set of automated test cases. Even though the Extreme Programming development paradigm includes the creation of a strong automated test suite, it is well known that tests are usually not complete, i.e., they do not cover all the possible program execution path combinations and all inputs [6, 39]. Furthermore, in a large scale software system, running all the tests potentially takes a considerable amount of time, which makes this approach less appealing for interactive usage.

3.4 Alternative Solution Approaches

The previous subsections outlined the techniques described in other papers together with their strengths and drawbacks. In the following, this section presents approaches for which no paper could be found in the context of the Smalltalk programming language.

3.4.1 Showing the Changes Before Execution

One approach for supporting the developer in a situation where she needs to rename an overloaded method is to inform her that there are multiple implementors. This idea is realised for the rename method refactoring in VisualWorks 7.9. The refactoring tool gives the option to show the changes before applying them. This allows to check whether there are implementors of methods with the same name that the developer does not want to rename. However, if the developer does not intend to rename all of the proposed methods, he must manually decide, which senders and implementors are to be changed, which brings him back to the questions described in Section 2.4.

3.4.2 Reducing the Refactoring Scope

An extension of the idea described above is to reduce the scope of a refactoring. The effect is that only senders and implementors belonging to a certain subset of methods in the source code are considered. Even though this feature is not available in VisualWorks up to version 7.9, all the necessary building blocks are present, which makes its implementation technically trivial. However, the decision how to partition the code must be taken by the developer, which limits this idea to systems where modules or layers are structured well enough to clearly identify the scope of the refactorings.

3.4.3 Dynamic Type Inference While Running a Productive System

Apart from dynamic type inference based on test cases described in Section 3.3.4, there is the possibility to gather type information from a running system used in production or a similar environment. Such a dynamic inference extracts the type information directly from a system that is used in a representative way rather than relying on ”good enough” test coverage. Potential drawbacks are slow performance or other disturbances of the productive system caused by the measuring mechanism. In addition, the type information is related only to deployed code, which causes a delay between development and availability of type information.

3.4.4 Extracting Type Information from the Database

Similar to dynamically inferring types while running a productive system is the idea of extracting type information regarding the instance variables of persistent objects from a database. Besides the advantages that no test cases are required and that the productive context is representative, this technique does not affect the performance of the productive system, which makes its application appealing. However, the database only provides type information regarding instance variables. Furthermore, it does not contain any type information for classes whose instances are transient.

This paper presents a different approach by proposing the use of a type inference mechanism, based on static code analysis. This technique avoids the shortcomings of the approaches mentioned before: no type declaration by the programmer is needed. Furthermore, the code analyser does not rely on the existence of a representative set of automated tests. Since type information needs to be precise for refactoring safely, accuracy is an important aspect of the inference mechanism presented in this paper. The upcoming subsections explain the solution approach and the details regarding its components.

3.5 Chosen Solution Approach

The creation of the polymorphic graph relies on type information extracted by a static code analysis. Section 3.7 outlines the structural building blocks of a method’s source code and their direct impacts on the type extraction. Based on the those building blocks, Section 3.8 focuses on the challenges associated with dependencies between methods.

Once the type information is extracted through the static code analysis, the polymorphic graph is built as explained in Section 3.11. Considering the extracted type information allows to remove edges and thereby potentially renders the graph incomplete. Consequently, as explained in Section 3.12, possible clusters emerge in the graph, which distinguish the different semantics of a single method name.

The next subsection introduces the concepts used for the type extraction.

3.6 Definitions of Concepts

Type A type is a set of classes representing all the possible values for a specific variable or expression.

The type information is extracted performing a static code analysis. This code analysis requires the source code represented as a parse tree with semantics attached to the various tokens in the source code. As indicated in Figure 3, in VisualWorks Smalltalk the parse tree is created using the existing refactoring framework [29]. The existence of this framework eases the implementation of the static code analysis.

Symbolic evaluation The static code analysis of a method is done by analysing the parse tree's components in the order they would be executed if the corresponding method was actually run. The impact of each single parse tree element or statement is considered, as well as each possible execution branch.

On the one hand, the symbolic evaluation provides type information when analysing an assignment. One example is the assignment to the instance variable `name` in the setter method `Movie>>name:`, shown in Listing 7. In the sample code snippet in Listing 8, the argument passed to the setter method `name:` adds the class `Translation` as one possible class candidate to the type of the instance variable `name` in the class `Movie`.

Listing 7. The assignment provides type information

```
name: aTranslation
 name := aTranslation
```

Listing 8. The argument determines the type of the instance variable

```
| theGodfather title |

title := Translation new.
title english: 'The Godfather'.
title german: 'Der Pate'.

theGodfather := Movie new.
theGodfather name: title.
theGodfather price: 3.
```

On the other hand, the static code analysis potentially needs type information. This is the case, when analysing both the side effects and the returned value of a message send, since the code analyser must know the possible classes of the receiver to consider only the relevant implementors of the sent message. For instance, when evaluating the method `Rental>>printTotalOn:`, shown in Listing 9, the code analyser requires the type of the variable `customer` to decide which implementor of the method `name` to evaluate.

Listing 9. The type of the variable `customer` is crucial for choosing the implementation of the method `name`

```
printTotalOn: aStream
  aStream nextPutAll: 'Customer name: '.
  aStream nextPutAll: customer name.
  aStream cr.
  aStream nextPutAll: 'Movie title: '.
  aStream nextPutAll: (movie name displayIn: customer
      language).
  aStream cr.
  aStream printNumber: self price.
  aStream nextPutAll: ' EUR'.
```

This dependency is a challenge when it comes to extracting the type information, because inferring the type for one program element potentially requires the type information of many other program elements.

The following two concepts, which are relevant during the static code analysis phase, emphasise this dependency.

Closed context inference It is a type inference mechanism which uses the symbolic evaluation defined in Section 3.6 at the level of a single method that does not depend on program state scoped outside the method. All the necessary information is either available or defined in the method itself. Listing 10 shows a method that satisfies those constraints.

Listing 10. The temporary variable does not depend on outer context

```
planetEarth
  | earth |
  earth := Planet new.
  earth name: 'Earth'.
  ^earth
```

Since the symbolic evaluation handles global variables differently than instance variables and method arguments, as will be described in Section 3.8.4, global variables are the only allowed exceptions among the variables scoped outside the method that are considered during closed context inference.

Open context inference It is an extension of the closed context inference. As soon as the symbolic evaluation encounters dependencies to variables or program state scoped outside the method, it interrupts the current method analysis to determine first the type of those variables. Since instance variables, method arguments, and block closure arguments depend on program state scoped outside the method, they trigger the open context inference. One example is shown in Listing 11

Listing 11. The type of argument `myName` depends on the sender of the method

```
planetNamed: myName
  | tmp |
  tmp := Planet new.
  tmp name: myName.
  ^tmp
```

Also the symbolic evaluation of the method `printTotalOn:`, which is shown in Listing 9, requires open context inference: the instance variable `customer` is defined outside the method's scope. Thus, as a prerequisite to symbolically evaluating the method, the type of `customer` has to be inferred.

The next subsection describes in detail the structure of the parse tree in Smalltalk and the semantics of the different kinds of elements contained in it, as well as the implications on the static code analysis with respect to extracting type information during the symbolic evaluation.

3.7 Building Blocks of a Program

According to the original Smalltalk 80 language described by Goldberg [14], the different elements in the parse tree are:

1. method signatures
2. calls to primitives
3. variable declarations
4. literal objects or expressions
5. message sends
6. variable assignments
7. returns of values or expressions

The latter three parse tree elements are composed of different sub-trees. This implies a dependency between them and the sub-trees. As an example, Figure 9 depicts the dependency graph for the parse tree of the method `price` shown in Listing 12.

Listing 12. The method is composed by various parse tree elements

```
price
  ^movie price * numberOfDays
```

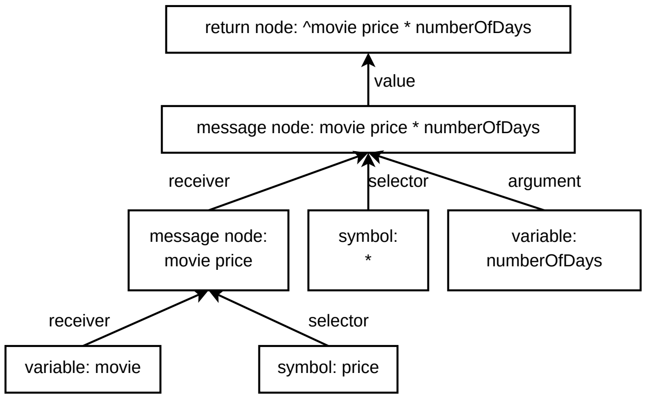

return node: ^movie price * numberOfDays
value
message node: movie price * numberOfDays
receiver
selector
argument
message node: movie price
symbol: *
variable: numberOfDays
receiver
selector
variable: movie
symbol: price

Figure 9. There are dependencies in the parse tree elements of the code shown in Listing 12

As a consequence of the dependency, the symbolic evaluation is recursive when performing the static code analysis.

The upcoming subsections contain explanations for each of the different code elements regarding their syntax by showing a sample code snippet in a shaded box, followed by a description of their semantics and the mechanism used during the analysis.

3.7.1 Method Signatures

The first line of code inside every method is the method's signature. Its purpose is to give the method a name and to specify the number of input arguments as well as their names. As the method signature is not executable code, during symbolic evaluation, only the argument names are considered as declarations. The extraction of the types of the arguments is further explained in Section 3.8.1.

3.7.2 Calls to Primitives

<primitive: 32>

Even though most of the Smalltalk programming language is written in itself, primitives are necessary building blocks for the Smalltalk system. They represent the technical interface to the underlying core of the virtual machine, which is written in a low level programming language.

By using primitives, the system can perform basic operations such as number arithmetic, number comparison, and I/O operations. The original Smalltalk 80 system featured only a small number of primitives to make most of the system available as Smalltalk source code and, thus, extensible. The further evolution of Smalltalk systems and their usage in productive environments demanded better performance, which was often achieved by adding new primitives. Even though in today's Smalltalk systems there are hundreds of primitives, they are part of the code elements for which the analysis is trivial. One reason is that most of them will not have an effect on the state of the variables in the system. Furthermore, primitives represent the boundary between the object oriented world and the low level libraries composing the virtual machine. Therefore, polymorphism is not available and the type of input and output parameters and values must be strictly specified. Accordingly, the code analyser presented in this paper considers the type of arguments and return values for primitives when extracting type information.

3.7.3 Variable Declarations

| temp |

The developer can declare variables in different contexts with different scopes:

- instance variables
- class variables
- method arguments
- block closure arguments
- temporary variables within methods or block closures

Since variable declarations do not represent executable code, no symbolic evaluation is applied. However, the code analyser adds the declared variable identifiers to the variable scope. The inferred type is initialised to `UndefinedObject`, which is the class of `nil`. The type for each variable is updated during the symbolic evaluation of the code contained in the variable's scope. A detailed explanation of the different kinds of variables follows in Section 3.8.

3.7.4 Literal Objects or Expressions

The Smalltalk language specifies literal objects or expressions and differentiates between pseudo-variables, literal expressions or values, and declared variable identifiers. Among the pseudo-variables there are `nil`, `self`, `super`, `true`, and `false`. In contrast to pseudo-variables, there are literal expressions and values, which create commonly used types, i.e., strings, symbols, numbers, literal arrays, and block closures. Performing the analysis of those code elements is straightforward, mainly due to two reasons.

Firstly, the class of the result of almost all such literals is known at compile time. The only exception is the pseudo-variable `self`, which could be of different classes if the implementing class has subclasses. However, even the class of `self` can be easily found, given the information about the receiver.

Secondly, the execution of those literal code elements will not trigger any side effect on the state of any variable. Hence, the code analyser focuses solely on the resulting value.

In contrast to the pseudo-variables, the type of the declared variables cannot be determined at compile time. Whenever the code analyser performs the symbolic evaluation of a variable, it looks up for the nearest scope declaring the underlying identifier, as does the virtual machine during the execution. Once found, the variable's scope and its type is extracted as described in Section 3.8.

The literal block closure triggers the creation of a special type candidate containing the block's source code and knowledge about its creation context. This information is needed when symbolically evaluating the block as explained in Section 3.8.5.

3.7.5 Message Sends

```
1 isPrime
2 isBetween: 1 and: 3
1 + 2
```

To perform the static code analysis of message sends, the following two steps have to be carried out in compliance with the order at run-time.

First, the code analyser begins with the evaluation of the arguments to determine their types. Since each of them could be a composite expression, the symbolic evaluation is recursive.

Second, the type of the message receiver is determined through symbolic evaluation of the corresponding parse tree expression. All the classes contained in the receiver's type are considered when looking up the implementation of the sent message.

For the method look-up, the code analyser follows the same rules as the virtual machine when doing the late binding during the execution: it looks for implementations of the sent message in the class hierarchy of the receiver. Unless the receiver is the pseudo-variable `super`, it considers the implementation that is nearest to the receiver's class. The mechanism regarding the method look-up corresponds to the implementation of the late-bound-polymorphism in Java and C♯ [15, 25]. Figure 10 illustrates the method look-up: when symbolically evaluating the sent message `foo` for a receiver of class `B`, the implementation in class `B` is considered. In contrast, the message `foo` sent to an instance of class `C` refers to the implementation in the superclass `A`.

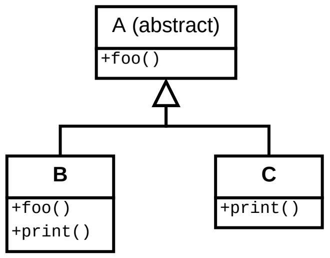


Figure 10. The symbolic evaluation takes the nearest implementation

While the classes that do not provide an implementation for the sent message are ignored, the implementation found in all other classes is symbolically evaluated. Considering all the implementors aims at analysing all the possible program flows to determine both their potential impact on the state of variables and the type of the return values.

3.7.6 Variable Assignments

```
temp := 1
```

When performing a symbolic evaluation on this kind of code element, two different aspects have to be considered. Firstly, an assignment might add one new class candidate to the type of the underlying variable. Secondly, the right hand side of the assignment has to be symbolically evaluated. This symbolic evaluation is recursive, since the assigned value might be a nested expression composed by other code elements.

Depending on the structure of the assigned expression, the actual evaluation at run-time of the right hand side of an assignment may trigger side effects on the program state. Given the implementation of `Stack>>pop`, presented in Listing 13, the assignment in the code snippet shown below changes the program state.

```
nextElement := stack pop.
```

By symbolically evaluating the right hand side of the assignment, the code analyser keeps track of potential side effects on the program state i.e., on the types of variables.

Listing 13. Sending the message `pop` changes the program state

```
pop
  | result |
  result := elements last.
  elements remove: result.
  ^result
```

Furthermore, according to the run-time behaviour, the result of the symbolic evaluation of an assignment is the type of the assigned value: Consequently, in the code snippet shown below, the type of the variable `y` is the type of the result of the assignment to the variable `x`, which is the assigned number.

```
y := x := 1
```

3.7.7 Returns of Values or Expressions

```
^name
```

Every method in Smalltalk returns a value, which is either determined by one or more explicit return statements or is the receiver itself, if the method does not include any explicit return statement. As a consequence of potentially having more than one explicit return statement, a method's return type is the union of the classes of all the potentially returned values.

The return program element terminates the execution of the method. It will not have an impact on any variable's state, unless the returned value is an expression whose evaluation triggers side effects on the state of variables.

This subsection explained the different types of parse tree elements, their dependencies and in which order the symbolic evaluation considers them. The next subsection gives a more detailed insight about when type information is needed and extracted and the challenges associated with the type extraction during the symbolic evaluation.

3.8 Extracting Type Information

As mentioned in Section 3.6, the code analyser extracts the types of variables and expressions during the symbolic evaluation. It starts with closed context inference, which performs type extraction for each parse tree element described in the previous subsection. However, as soon as the code analyser encounters a variable scoped outside the method, the closed context inference is suspended to determine the type of the variable. Subsequently, the closed context inference proceeds using the type information extracted during the open context inference.

For determining the type of each different kind of variable scoped outside the method, the open context inference applies different strategies, which are described in the following subsections.

3.8.1 Method Arguments

First, the code analyser finds all the senders to the name of the underlying method. This step is part of the VisualWorks refactory browser. Second, all the sender nodes are symbolically evaluated according to the closed context inference mechanism, keeping track of the types of all the variables and expressions used in the method. Third, for all the sender nodes whose receiver type contains the method implementor for which the method argument has to be inferred, the type information for the argument is considered.

For instance, inferring the type of the argument in the method `Movie>>name:`, presented in Listing 14, requires the symbolic evaluation of the method that contains the senders of `name:`, which is shown in Listing 15. Among the two emphasised senders, only one is relevant for the argument in `Movie>>name:`, because only one refers to a receiver of type `Movie`. The symbolic evaluation of the expression passed as an argument reveals that the class `Translation` is the type candidate for the method argument.

Listing 14. The type of the argument depends on the senders

```
name: aTranslation
  name := aTranslation
```

Listing 15. Only one sender of the method `name:` is relevant

```
title := Translation new.
title english: 'The Godfather'.
title german: 'Der Pate'.

theGodfather := Movie new.
theGodfather name: title.
theGodfather price: 3.

alice := Customer new.
alice name: 'Alice'.
alice language: 'english'.
```

3.8.2 Instance Variables

The code analyser first considers all the writers of the particular instance variable. A writer is a method that contains an assignment to a specific variable. Constructors and setter methods are among the writers. While symbolically evaluating all the writers, the code analyser keeps track of the class candidates assigned to the instance variable. The type of the assigned value might depend on a method argument. In this case, for first determining the types of the arguments, the mechanism described in Section 3.8.1 is used.

The symbolic evaluation of writer methods reveals the class candidates associated with a particular instance variable. However, it does not provide any information regarding the internal state of the object contained in the instance variable.

Therefore, additional analysis is needed to enrich the type information concerning the state of the instance variable. This additional analysis consists of symbolically evaluating all the variable's readers, because any message send to it can alter its internal state. A variable's reader is a method that is directly accessing an instance variable. As a consequence, for further analysing the state of the instance variable, all readers are symbolically evaluated.

Typical readers are the getter methods, which give access to the variable to other methods in the program. Hence, methods referring to getters potentially modify the variable's internal state. For this reason, all references to the variable's getter are symbolically evaluated as well, with the objective of collecting more type information about the variable's internal state.

3.8.3 Indexable Variables

Collections use indexable variables. There are collections, such as strings and symbols, that can contain only specific types of elements. This constraint is enforced by the system. Thus, the symbolic evaluation does not perform any analysis on collections that have built-in type constraints.

However, generic collection classes allow the programmer to add any kind of object as element. The code analyser performs type inference on the elements stored as indexable variables, as they are part of the internal state of an object. The main difference between indexable variables and instance variables is that there is not a separate getter or setter for each of them. In contrast, there are only a few primitives granting access for reading, adding, or removing elements. The symbolic evaluation keeps track of the elements. Initially, when the collection is instantiated, it does not contain elements and therefore no type is associated with its indexable variable. Whenever an element is added through the aforementioned primitives, the symbolic evaluation first determines the type of the object passed as an argument. Then its type is considered as a type of the indexable variable.

As there is only one common accessor for reading elements of the collection, the index that is passed as an argument decides which element is returned. When extracting type information, the code analyser keeps track only of the class candidates of variables and expressions, rather than their specific values. Hence, the symbolic evaluation infers that the index passed as an argument to the accessor is an integer. However, it does not infer its exact integer value. Consequently, it is unable to infer which element of the collection is returned. For this reason, when fetching an element of the collection, shown in Listing 16, the symbolic evaluation considers the union of all the types of the elements in the collection as return type, which are the classes `Integer` and `String`.

Listing 16. The code analyser considers all element types as potential return type when fetching an element

```
collection := OrderedCollection new.
collection add: 5.
collection add: 'test'.
^collection at: 1
```

As mentioned in Section 3.6, when analysing the statements and expressions within a method, the symbolic evaluation maintains the actual run-time execution order. However, it is unable to maintain the program flow in general, since the order in which methods are called depends on the program state, external events, and user input. For this reason, it is impossible for the static code analyser to infer the order in which elements are added, removed, or read. Listing 17 illustrates an example of a collection that is modified in different methods. The element returned by the method `getFirstElement` depends on whether the method `removeSomeElements` is called first.

Listing 17. The order in which the methods are called determine the content of the collection

```
initialize
  collection := OrderedCollection new.
  collection add: 'test'
  collection add: 1.

removeSomeElements
  collection remove: 'test'

getFirstElement
  ^collection at: 1
```

Consequently, for the symbolic evaluation, removing elements from a collection has no impact on the type of its elements and on the return type of methods that read single collection elements: no type candidate is removed. The types of all the elements added to the collection are always considered, since the code analyser can not determine whether a particular element has been removed before accessing the collection's content.

3.8.4 Class Variables and Global Variables

In object oriented programming, the usage of class variables and global variables is considered a bad practise, unless used in the context of design patterns, e.g., the Singleton pattern [13, 26]. Within the Smalltalk system, global variables are used in special contexts, such as system configuration. Usually, the content of global variables is not changed during the program execution, since they are reserved for configuration. Some of the global variables are changed manually, which makes it impossible to analyse what class candidates might be assigned to them. Global variables and class variables are accessible at system level. For this reason, the code analyser accesses their value to directly extract the class candidates. To avoid changes to be made to their content during the symbolic evaluation, the type inference process is synchronous, i.e., modal. This means that the user is not able to make any changes to the source code or to the global variables until the type extraction is completed.

3.8.5 Block Closures

Block closures are best described as anonymous methods that contain executable code, including their own arguments and temporary variables. Nevertheless, they have to be treated differently when inferring the types of variables used inside block closures.

It is crucial to distinguish between the instantiation of a block and its execution. Once instantiated, a block might be stored in a variable or passed as argument. Sending particular messages that invoke certain primitives, such as `value` and `value:`, trigger its execution.

As mentioned in Section 3.7.4, every time the symbolic evaluation encounters the creation of a block closure by a literal parse tree element, it creates a type candidate that contains the entire code contained in the block closure as well as a reference to the creation context. This information is used as follows during the symbolic evaluation and type extraction within blocks.

Temporary variables defined inside the scope of the closure itself are handled according to the closed context inference concept defined in Section 3.6. Block closures may also contain bindings to variables defined outside their own scope, such as instance variables. In this case, their type is extracted as described in Section 3.8.2. However, there is an important detail regarding the binding of variables in block closures: the block might be evaluated in a different context than the context where it has been created. Corresponding to the semantics of blocks at run-time, the binding of variables scoped outside the block is done based on the creation context of the block closure. Thus, the code analysis on such variables is performed in the creation context of the block closure.

3.8.6 Block Closure Arguments

Even though they are syntactically different from method arguments, block arguments are similar to them. The only difference is that, instead of having a user defined method name with which the arguments are passed, there is a small set of generic methods for evaluating block closures and passing the corresponding parameters.

Because the method name for the execution of a block is generic, there is no support for finding the references to a specific block as for method names. This makes it impossible to use the mechanism for inferring the type of method arguments and, thus, demands for a different strategy: according to the explanation given in Section 3.7.4, when encountering the creation of a block closure, the code analyser creates a type candidate that contains the code of the block and a reference to its creation context. The resulting type might be either stored in a variable or passed as an argument, like any other type candidate. As soon as the block type candidate is identified as potential receiver of a message during the symbolic evaluation of a message send, as described in Section 3.7.5, the sent message potentially triggers the evaluation of the code contained in the block. The method lookup for block closures during the code analysis is the same as for other type candidates. Hence, the decision depends on whether the implementation of the sent message found in the hierarchy of the system class `BlockClosure` contains calls to certain primitives. If the method implementation triggers the evaluation of the block, the expressions passed as arguments to the block are symbolically evaluated and their type is associated with the block arguments.

In contrast to the evaluation of externally defined variables used within the block, the symbolic evaluation of the expressions passed as arguments takes place in the method context where the block is evaluated rather than where it was created. This interpretation complies with the behaviour of block closures at run-time.

The following sample code passes a block that expects the argument `amount` to the method `Invoice>>printOn:unless:`. The type of the block argument `amount` corresponds to the type of the variable `accountBalance` in `Invoice`, which is passed as argument when triggering the evaluation of the block by sending the message `value:`.

Listing 18. The variable `accountBalance` determines the type of the block argument

```
Printer>>processInvoice: anInvoice
    anInvoice printOn: self unless: [:amount | amount = 0]

Invoice>>printOn: aPrinter unless: aBlock
    (aBlock value: accountBalance) ifFalse: [aPrinter print:
      self]
```

3.9 Not Considered Code Constructs

As mentioned in the previous subsections, the type inference is based on static code analysis. As Smalltalk supports reflection - which means that the programmer can dynamically query and change the program's structure, implementation, and state [11] - not necessarily all the relevant code is available at compile-time.

Two common contexts where reflective features are used are code generation or data binding between the model and the GUI. For some of the reflective code constructs, such as the ones shown below, the argument values such as method names are available at compile time.

- anObject perform: #foo
- anObject perform: 'foo' asSymbol
- anObject perform: ('foo' , ':') asSybmol

This means that all the information that is necessary for performing the static type inference is present at compile time. However, the prototype implementation of the type inference mechanism presented in this paper does not consider code that uses any of the reflective features. Consequently, it does not extract type information that results from reflective code. The reason for not considering the subset of reflective code constructs that provides the necessary information at compile-time is that the presented code analyser is still a prototype rather than being a sophisticated and mature tool.

In contrast to the examples shown above, if the code constructs, such as message names, method implementations, or even classes, depend on external state or input, as shown in Listing 19, they are not yet present at compile-time. As a result, the static code analyser is unable to consider its implications with respect to types and program state.

Listing 19. The method to be called depends on the content of the file

```
callMethodOn: anObject
    | methodName |
    methodName := 'methodName.txt' asFilename
      readStream nextLine.
    ^anObject perform: methodName asSymbol.
```

3.10 Sample Type Analysis

The aforementioned subsections described the details concerning the static code analysis and the strategies used for extracting the type information depending on the program structure. This subsection illustrates the type analysis step by step for the sample refactoring of renaming the method `Customer>>name` to `Customer>>lastname` mentioned in Section 2.5.

The purpose of the refactoring is to rename the method `name` in the class `Customer`. The method `name` in the sample code has two implementors: `Movie` (Listing 1) and `Customer` (Listing 2). The sample code contains two senders of the method `name`, both in the method `printTotalOn:` and emphasised in Listing 20. According to the implementation of the refactory framework [29], the program elements sending a message are named "message node" or "message sends". They do not represent the whole method that contains the message send, instead, they represent only the single parse tree element. This distinction is crucial, since one method can contain more than one message send with the same name as shown in Listing 20.

Listing 20. Knowing the type of both receivers of the message `name` is necessary

```
printTotalOn: aStream
  aStream nextPutAll: 'Customer name: '.
  aStream nextPutAll: customer name.
  aStream cr.
  aStream nextPutAll: 'Movie title: '.
  aStream nextPutAll: movie name.
  aStream cr.
  aStream printNumber: self price.
  aStream nextPutAll: ' EUR'.
```

Understanding the dependencies between the senders and the implementors is crucial for constructing the polymorphic graph and, thus, for safely renaming the method. The following description focuses on the single steps that are necessary for extracting the type information for the senders of the method `name` in the given sample program, which is a prerequisite for creating the polymorphic graph.

For both message nodes emphasised in the aforementioned code listing, the code analyser determines the receiver's type using symbolic evaluation on the method `printTotalOn:`. As soon as the symbolic evaluation encounters the message node customer name, it interrupts the evaluation of `printTotalOn:` for first identifying the type of the receiver, which is the instance variable `customer`. Similarly, it infers the type of the instance variable `movie` to understand whether the message node movie name is relevant for the rename method refactoring. According to the description in Section 3.8.2, it is necessary to consider the instance variable's writers, which, in the case of the class `Rental`, are the setter methods `customer:` and `movie:` shown in Listing 21.

Listing 21. the types of the variables depend on the arguments

```
customer: aCustomer
  customer := aCustomer

movie: aMovie
  movie := aMovie
```

When symbolically evaluating each of the setters, the code analyser detects the dependency on the method argument and determines first its type. According to the mechanism presented in Section 3.8.1, this requires the symbolic evaluation of the methods calling the setter method. The code fragment illustrated in Listing 22, which creates the instances in the sample program, contains the only callers to the setter methods `customer:` and `movie:`.

Listing 22. Extracting the types of the arguments passed to the setter methods `movie:` and `customer:` from the callers

```
| theGodfather alice rental stream title |

stream := WriteStream on: String new.

title := Translation new.
title english: 'The Godfather'.
title german: 'Der Pate'.

theGodfather := Movie new.
theGodfather name: title.
theGodfather price: 3.

alice := Customer new.
alice name: 'Alice'.
alice language: 'english'.

rental := Rental new.
rental movie: theGodfather.
rental customer: alice.
rental numberOfDays: 2.

rental printTotalOn: stream.
stream contents.
```

The two senders pass as arguments the temporary variables `theGodfather` and `alice` respectively. By examining the two assignments emphasised in Listing 22, the symbolic evaluation infers their types: The type of `theGodfather` contains only `Movie` as class candidate while `alice` is of type `Customer`. Putting the type information of the previous steps together, the symbolic evaluation shows that, in the method `printTotalOn:`, the message send of `name` in `customer name` refers to the implementation in the class `Customer` while `movie name` refers to the implementor `Movie`.

The next subsection focuses on building the polymorphic graph, based on the extracted type information for the method `name`.

3.11 Creating the Polymorphic Graph

According to the definition given in Section 3.2, the polymorphic graph consists of all the program elements sending a message with one particular name. Each message node is represented as a vertex of the polymorphic graph's right hand side. For all the message node vertices, the code analyser identifies the class candidates of the receiver as described in the previous subsections. All the receiver's class candidates are vertices on the left hand side of the polymorphic graph. The graph contains edges connecting each message node with all of its receiver's class candidates.

Figure 11 depicts the resulting polymorphic graph, which uses the extracted type information for the method `name` in the sample program.

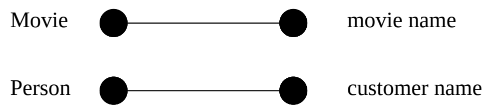


Figure 11. Edges represent the type information for the method `name`

This graph clearly shows two clusters: `movie name` referring to `Movie` and `customer name` referring to `Person`. The next subsection describes the algorithm for identifying the separate clusters in the graph.

3.12 Separating Clusters in the Polymorphic Graph

The algorithm used for identifying and separating the individual clusters within the polymorphic graph is described below. The corresponding pseudo-code-listing can be found in the thesis that builds the base for this paper [33].

Initially, none of the vertices is assigned to a specific cluster. All of them belong to the list of unassigned vertices. The program creates a new empty cluster assigning one of the sender vertices to it, removing it from the list of the unassigned sender vertices. All the implementor vertices that are directly connected to it are added to the same cluster. All the sender vertices that are directly connected to any of the previously added implementor vertices and are not yet part of the cluster are also assigned to the same cluster. The program adds the implementor vertices that are connected to the newly added sender vertices to the cluster unless they already belong to the cluster. This mechanism of adding connected sender and implementor vertices continues until no more new vertices are added. If there are still unassigned sender vertices left, the entire procedure is run again on the remaining list of unassigned vertices of the graph, creating one cluster at a time.

Refactorings involving overloaded method names rely on the resulting clustered polymorphic graph. The following subsection presents a sample refactoring.

3.13 Mechanisms of Typed Refactorings

As a concrete example for a refactoring that uses type information, this paper focuses on the widely used ”Rename method” refactoring. The reason for this choice is that it can be used as a stepping stone for performing other refactorings on overloaded method names by first renaming the relevant method to a unique name, and then performing other existing refactorings. Nevertheless, the type information allows to implement enhanced versions of other refactorings that benefit from type information, such as the refactorings mentioned below:

- add or remove method argument

  As for the ”rename method” refactoring, the type information helps to limit the scope of the refactoring.
- inline method from component

  If the receiver type is known, the candidate list of method implementations to inline can potentially be reduced.
- extract method to component

  If the receiver type is known, the set of target classes to which to extract the code can be more precise.
- finding the senders or implementors

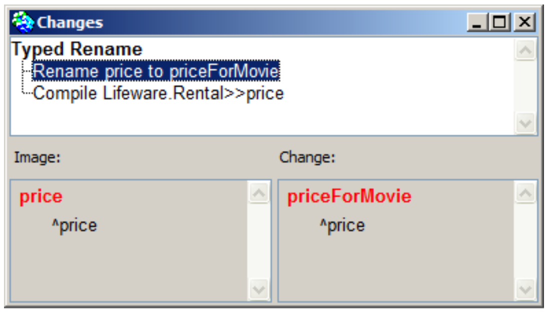
  As described in Section 2.4, finding the references or implementors is crucial when performing manual refactorings. The clusters built on the extracted type information allow to deliver more precise results in a code base with overloaded method names by listing only the senders or implementors in one particular cluster.

As mentioned in Section 3.6, in Smalltalk there is a refactoring framework available, which provides the interface for extending existing refactorings or implementing new ones. This framework provides the possibility to use code rewrite rules, which apply a pattern matching mechanism for both finding and changing specific structures in the source code, such as references to methods.

The prototype of the typed rename method refactoring uses the clusters in the polymorphic graph to avoid renaming all the references to the target method name by changing only the references that are contained as vertices in one cluster. As for the built-in rename method refactoring, the user has to choose the method to rename by selecting a specific implementor. The scope of the typed rename method refactoring is the cluster that contains the class of the chosen implementor among its vertices.

The implementation of the typed rename method refactoring uses the built-in refactory framework by applying a customised rewrite rule for performing the pattern matching within the parse tree. While doing the pattern matching, every node in the parse tree is visited. The pattern matching rule first checks whether the currently analysed node is a message node sending the message to be renamed. If this is the case, it also checks whether the message node is among the sender vertices of the relevant cluster, which has been chosen as a target scope for the rename. Only if both conditions are met, the refactoring changes the name of the sent message in the visited parse tree element.

The built-in refactoring framework allows to visualise the changes before their actual execution, as shown in Figure 12 and Figure 13. This is a crucial step for evaluating the tool’s utility and to make the developer familiar with the mechanisms of this particular refactoring. The purpose is to speed up acceptance and the trust building process within the development team, by giving them full control over the proposed changes. The changes are undoable after their confirmation through the built-in undo menu.

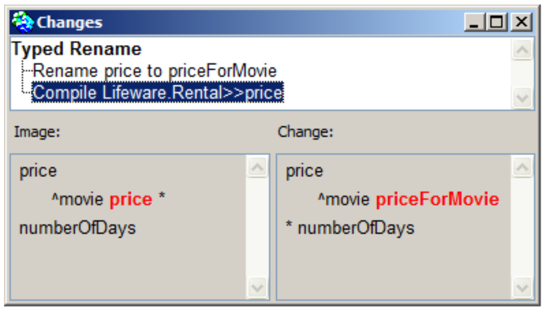


Typed Rename
Rename price to priceForMovie
Compile Lifeware.Rental>>price

```
price
    ^price
```

```
priceForMovie
    ^price
```

Figure 12. The implementor is renamed


Changes

Typed Rename

Rename price to priceForMovie

Compile Lifeware.Rental>>price

Image:

```
price
    ^movie price *
numberOfDays
```

Change:

```
price
    ^movie priceForMovie
* numberOfDays
```

Figure 13. The user can see which senders will be changed

This concludes the step-by-step explanation of the different phases and concepts of the implementation of the sample refactoring.

3.14 Limitations

The solution approach and its implementation are subject to the following limitations.

3.14.1 Use of Reflection

The Smalltalk programming language offers powerful reflection features. A reflective programming language offers the possibility to dynamically query and change the program's structure, implementation, and state [11].

In Smalltalk, reflection provides also the possibility to intercept method invocations to alter the mechanism of the method look-up [11]. During the symbolic evaluation, the static code analyser assumes the standard method look-up for sent messages, as explained in Section 3.7.5. As a consequence, the symbolic evaluation does not consider potential alterations of the method look-up.

Furthermore, as mentioned in Section 3.9, the code analyser does not consider code that uses reflective features when extracting type information.

3.14.2 Absence of Formal Model

The symbolic evaluation considers the characteristics and the side effects of each of the possible parse tree elements to extract type information, as explained in Section 3.7 and Section 3.8. More than 250 test cases informally verify the type information extracted during the static code analysis. However, its implementation is not based on a formal model. Consequently, there is no formal proof for completeness or correctness of the resulting polymorphic graph.

3.14.3 Slow Performance

For some program elements, extracting type information requires analysing many methods of the program's source code. In particular, the extraction of type information for instance variables potentially triggers the symbolic evaluation of hundreds of methods, because of the dependencies mentioned in Section 3.8.2. The time required to analyse the large number of methods potentially hinders the interactive usage of the prototype of the typed refactoring.

The next section illustrates statistics that consider the time required for extracting type information on individual methods to assess the effectiveness of the prototype.

4. Evaluation

This section assesses the potential contribution of the implemented prototype from the user's perspective. It defines research questions and explains the strategy for answering them by giving a description of the experimental setup. It presents the results coming from a statistical analysis and discusses the limitations of the experiment.

4.1 Research Questions

When considering the adoption of the implemented prototype of the refactoring tool, the developer is interested in the following questions:

Q1 Is the tool capable of supporting every case where she wants to rename overloaded methods?

Q2 If not, what is the percentage of cases, where it can be used?

Q3 Is the tool fast enough to be used interactively?

The upcoming subsections present an experiment and the according statistical analysis for giving an estimation of the probability of successfully applying the implemented prototype. In addition, the experiment uses time limits to assess whether interactive usage is feasible.

4.2 Evaluation Approach

In the source code there exist overloaded methods that the prototype can successfully rename, and other overloaded methods that cannot be renamed by the prototype within a reasonable time. Even though the number of methods of both kinds is unknown, the outcome for each method is deterministic and is either success or failure. The probability of success when renaming one arbitrarily chosen overloaded method depends on the probability of choosing a method on which the prototype applies successfully. This situation corresponds to the concept of a Bernoulli trial with unknown probability of success [1, 10].

4.2.1 Experimental Design

The evaluation relies on an experiment based on repeating the Bernoulli trial for 2000 times, i.e., binomial experiment, to give an estimation for the unknown probability and to measure the influence of the time limit on it. The experiment consists in drawing statistical samples from the Smalltalk image and in measuring the outcome. Since extracting the polymorphic graph for a method represents the critical step for the enhanced rename method refactoring, the decision regarding success or failure for one trial on a single method, which corresponds to the Bernoulli trial mentioned before, is reduced to the question whether the code analyser can extract the polymorphic graph for the chosen method within the given time limit. The methods for which the code analyser could complete the analysis within the given time count as successes as opposed to the methods, where no result could be extracted within the given time.

According to the aforementioned textual explanation, the following definitions describe the experiment with statistical notations.

Population: methods having at least two implementors, i.e., overloaded methods

N: number of overloaded methods

M: number of overloaded methods, for which the type inferencer is able to extract the polymorphic graph within the given time

$X \sim B(1, p)$: Bernoulli trial with a success probability p, where:

X is the random variable, which assumes two possible values:

$$X = \begin{cases} 1 & \text{success} \\ 0 & \text{failure} \end{cases}$$

Success means choosing a method for which the code analyser can extract the polymorphic graph

$$P(X) = \begin{cases} p & X = 1 \\ 1 - p & X = 0 \end{cases}$$

In this experiment, p is the proportion of overloaded methods in the population, for which the code analyser can extract the polymorphic graph, i.e., M/N

Y: random variable for the repeated Bernoulli trial, i.e., binomial experiment, describing the number of successes in the sample

$Y \sim B(k, p)$: binomial experiment consisting of k Bernoulli trials with the probability p. The sample size k of the performed experiments is 2000

4.2.2 Scoping the Samples

The VisualWorks image of Lifeware is the target of the experiment. In addition to the code that is specific to each of Lifeware's customers, it contains framework code for the insurance domain, implemented by Lifeware. Even though the image has a monolithic structure, classes that are relevant only for one of the customers and, hence, do not belong to the framework code, are marked as such. Furthermore, customer-specific code is allowed only to depend either on code used for the same customer, or on framework code.

This structure of the image potentially facilitates the extraction of type information, because two method implementors belonging to customer-specific code of distinct customers are independent, unless the framework code contains a sender of the same method name. Such a sender in the framework code introduces a dependency between the distinct customer-specific code bases, since renaming one method possibly requires renaming the sender in the framework code and therefore also the implementor in the code of the other customer.

Figure 14 illustrates an example, where the framework code does not contain a sender of the method and, hence, the two customer-specific code bases are independent. In contrast, Figure 15 shows a class in the framework code that contains a sender of the relevant method name, introducing a dependency between the customer-specific code bases.

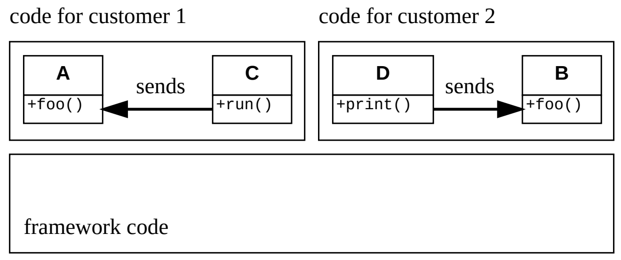


Figure 14. The customer-specific implementations are independent

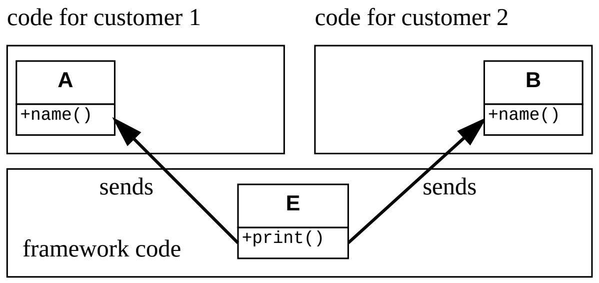


Figure 15. The sender in the framework introduces a dependency

For methods without senders in the framework code, such as shown in Figure 14, the code analyser considers the customer-specific parts as independent partitions and, therefore, ignores the implementors and senders in the code of the other customers. This potentially reduces the number of dependent methods that must be processed during the type inference for a specific program element. Consequently, the likelihood of successfully extracting the polymorphic graph within the given time might differ for method names that are implemented and sent only within customer-specific code compared with method names used also within the framework code or the Smalltalk base classes.

To quantify this difference, two independent sets of methods build the base of the experiment, each of which consists of 2000 randomly selected methods among those having at least two implementors chosen once from the entire Lifeware image, and once excluding methods that have either implementors or senders also in the framework code. The reason for distinguishing the two cases is to give the programmer a more precise estimation of the likelihood of successfully using the prototype of the enhanced rename method refactoring on any Smalltalk image.

4.2.3 Applying Time Limits

In addition to the partitioning, the applied time limit potentially influences the success rate of extracting the polymorphic graph. In order to assess whether increasing the time limit raises the number of methods for which the code analyser is capable of extracting the polymorphic graph, each method name has been processed with two different time limits, i.e., one and ten seconds, creating dependent pairs of samples [10]. The motivation for choosing those particular time limits is the interactive usage of the prototype, which is a crucial criteria for the acceptance of the tool among programmers [28].

4.2.4 Random Selection

The application of the model of a series of Bernoulli trails, as explained in Section 4.2.1, requires the single trials to be stochastically independent, which means that the result of one trial does not alter the probability of the outcomes of the following trials [1, 10]. In order to achieve stochastic independence for the creation of the samples of the 2000 methods, a chosen method was not excluded for the random choice of the next method, allowing a method to appear more than once in the sample. The evaluation presented in this paper also assumes that randomising in VisualWorks Smalltalk is stochastically independent and equally likely among all the available choices.

4.2.5 System Specification

Table 1 gives details regarding the specifications of the system used for the experiment.

CPU	Intel Core i5-2500K, 3.3 GHz
Main memory	8 GB
Operating system	Windows 7 Ultimate x64 SP 1
IDE	VisualWorks 7.41
Number of classes	37,700

Table 1. System specification

4.3 Statistical Analysis

Measuring the two independent sets of 2000 methods for each of the two time limits revealed the results depicted in Table 2.

scope	timeout (seconds)	# successes	% successes
global	1	505	25.25%
	10	507	25.35%
partitioned	1	876	43.80%
	10	878	43.90%

Table 2. Statistical result for the two paired experiments with 2000 samples

At the first glance, the success rate for methods taken from the entire image differs with respect to the success rate for methods taken only from the customer-specific partitions, while increasing the time limit seems to cause only marginal improvements.

However, deciding whether those first observations can be confirmed requires a statistical analysis. Therefore, for each of the four extracted samples illustrated in the table, the binomial proportion confidence interval is calculated. It provides information regarding the proportion of overloaded methods in the image for which the prototype could successfully extract the polymorphic graph within the given time limit. Since for a single Bernoulli trial, the probability of success is the proportion in the population, the confidence interval corresponds to the estimated probability of success for a single trial and, hence, for the rename of one randomly chosen method.

The calculation of the confidence interval is based on the widely used formula of Laplace [36]. It approximates the underlying binomial distribution with a normal distribution, which can be done if both the number of successes and the number of failures in the sample are greater than ten [10, 34]. Table 2 shows that the observed numbers are far greater than the required minimum for approximating the binomial experiment with a normal distribution.

The formula used for calculating the confidence interval is shown below,

$$\left(\overline{x} - z_{(1-\frac{\alpha}{2})}\frac{s}{\sqrt{n}}; \overline{x} + z_{(1-\frac{\alpha}{2})}\frac{s}{\sqrt{n}}\right)$$

where the meaning of the used symbols is as follows:

n: sample size

$\overline{x}$: proportion of observed successes in sample

s: standard deviation of the sample

$1-\alpha$: confidence level

$z_{(1-\frac{\alpha}{2})} = \Phi^{-1}(1-\frac{\alpha}{2})$: the value of the quantile function for the standard normal distribution for $1-\frac{\alpha}{2}$

Table 3 presents the results of the calculation with a confidence level of 99 % and an error level of 1 %, respectively.

		confidence interval for the probability	
scope	timeout	lower bound	upper bound
global	1	22.75%	27.75%
	10	22.84%	27.85%
partitioned	1	40.94%	46.66%
	10	41.04%	46.76%

Table 3. The confidence interval gives an estimation for the probability

After calculating the confidence interval for the two sample pairs, i.e., global and partitioned, each measured with two time limits, the next step is to assess the effect of increasing the time limit. For this purpose, Table 5 combines the two sample pairs, which are summarised in Figure 4, together as new samples.

The idea is to quantify the improvement triggered by increasing the time limit from one to ten seconds. Among the methods for which no type information could be extracted within one second, potentially exist candidates for which within ten seconds the code analyser was able to extract the polymorphic graph. In other words, the interesting aspects of the combined samples, presented in Table 5, are the number and the proportion of methods that transform from failures to successes by changing the time limit.

	timeout: 1 second		timeout: 10 seconds	
scope	successes	failures	successes	failures
global	505	1495	507	1493
partitioned	876	1124	878	1122

Table 4. The number of failures after 1 second is the sample size of the combined sample shown in Table 5

scope	failures with 1 second	yield with 10 seconds	yield rate
global	1495	2	0.13%
partitioned	1124	2	0.18%

Table 5. Increasing the time limit brings almost no improvement

The purpose of combining the two sample pairs is to calculate another confidence interval on the probability of successfully computing the polymorphic graph within ten seconds on methods for which within one second the code analyser could not extract type information. However, as Table 4 shows, increasing the time limit brought success solely for two methods out of 1495 and 1124 respectively. This corresponds to a success rate of far less than one percent. Actually, the number of additional successes is too small for applying the formula of the confidence interval, since the samples do not satisfy the previously mentioned conditions for approximating a binomial experiment with a normal distribution. Still, the numbers clearly indicate that increasing the time limit yields almost no additional value.

4.4 Results

Based on the aforementioned analysis, the answers to the research questions stated in Section 4.1 are the following:

Q1 Is the tool capable of supporting every case where the developer wants to rename overloaded methods?

A1 No, the tool is unable to support every case.

Q2 If not, what is the percentage of cases, where it can be used?

A2 The probability of success depends on the scope. If the method is only implemented and sent in customer-specific code, the estimated probability lies in the range between 41% and 47%, otherwise it is between 23% and 28%.

Q3 Is the tool fast enough to be used interactively?

A3 The estimated probabilities assume a time limit of one second, which is feasible for interactive usage. If the code analyser could not extract the type information for a specific method within one second, increasing the time limit to ten seconds rarely leads to success.

4.5 Threats to Validity

This subsection mentions the limitations of the evaluation approach and of the corresponding statistical analysis. Each of the upcoming subsections discusses in detail one of the limitations.

4.5.1 Considering Only the Lifeware Image

The evaluation and the statistical analysis presented in this section consider only one particular VisualWork Smalltalk image, which is the one used by Lifeware. From the company's foundation in 1998, starting with three developers, it evolved to an image containing more than 37,000 classes, maintained by 30 developers. Even though the image is the code base of a large scale industrial software project, it might not be representative and, hence, potentially inadequate to generalise statements or estimations that are based on the analysis. The estimated probability of success might vary significantly based on the project size, business domain, team size and other factors. The evaluation presented in this section does not consider those factors.

4.5.2 Not Verifying Correctness

As mentioned in Section 4.2.1, the measurement is reduced to the decision whether for a specific method the code analyser is capable of extracting the polymorphic graph within the specified time limit. If this is the case, the observation is considered as a success. Because of the big sample size - four experiments, each consisting of 2000 methods - manually verifying whether the extracted polymorphic graph is correct was not feasible. Consequently, the correctness of the extracted samples has not been verified. However, more than 250 test cases verify the soundness of the extracted type information and the corresponding polymorphic graph on selected examples.

4.5.3 Selecting Methods Randomly

The evaluation estimated the probability of success based upon a binomial experiment that prerequisites random selection for the extracted sample. Consequently, the estimated probability is only valid for random selection. However, rather than being the outcome of a random selection, the method a programmer needs to rename is driven by business requirements and software maintenance activities. For this reason, the probability of success during daily development potentially differs from the estimation.

4.5.4 Assuming Stochastic Independence on Random Generator

The necessary condition for applying the model of the binomial distribution is that the single trials satisfy the condition of the Bernoulli trial, i.e., the single trials need to be stochastically independent [1, 10]. Since the random choice of the 2000 methods uses the built-in random generator of Smalltalk, the evaluation assumes that it satisfies this condition.

5. Conclusions

This paper addresses the challenges with respect to refactoring automation and tool support in Smalltalk, triggered by the lack of type information at compile-time. In particular, this paper focuses on renaming overloaded methods that have the same name.

It proposes the usage of static code analysis to extract type information, which can be used by refactoring tools, explained in a detailed step-by-step manner. A prototype implementation serves as a proof of concept of the solution's approach. An experiment and its statistical analysis, aimed at assessing the utility of the prototype, estimate the success rate ranging from 23% to 47%.

5.1 Contributions

This paper presents a concrete proposal for performing type inference based on static code analysis. Using a modern hardware setup, it proved to be feasible in terms of response time, when applying it to a subset of methods.

This is an important result; given that, the research also reveals that refactoring tools in Smalltalk could greatly benefit from using type information.

Even though the evaluation demonstrates that the presented approach is not applicable for all the cases, there is a reasonable proportion of cases, where the prototype provides help for the developer. For this reason, Lifeware decided to use the prototype within the company.

5.2 Future Work

The static code analysis presented in this paper is not based on a formal model. Formalising the underlying solution approach would improve the safety of the type inference and, hence, make the prototype more trustworthy towards potential users among developers.

Additional analysis of the failures for identifying bottlenecks as well as optimisations for modern multi-core CPUs and use of sophisticated caching strategies could increase the success rate, which would make the presented refactoring tool more appealing for developers.

Further assessments of the prototype in the context of different projects could reveal whether the success rate depends on project-specific factors, such as the program's size or complexity.

Making the implemented prototype available to the Smalltalk community could stimulate interest, and thereby potentially increase the development and research resources for improving it.

References

[1] O. Anderson, W. Popp, M. Schaffranek, D. Steinmetz, and H. Stenger. *Schätzen und Testen: Eine Einführung in Wahrscheinlichkeitsrechnung und schließende Statistik*. Springer Berlin Heidelberg, 2nd edition, Apr. 1997.

[2] K. Beck. *Extreme programming explained : embrace change*. Addison-Wesley, Boston MA, 2nd edition, 2004.

[3] A. H. Borning and D. H. H. Ingalls. A type declaration and inference system for smalltalk. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 1982*, pages 133–141, Albuquerque, Mexico, 1982.

[4] G. Bracha and D. Griswold. Strongtalk. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications - OOPSLA 1993*, pages 215–230, Washington, D.C., United States, 1993.

[5] D. Campbell and M. Miller. Designing refactoring tools for developers. In *Proceedings of the 2nd Workshop on Refactoring Tools - WRT 2008*, pages 1–2, Nashville, Tennessee, 2008.

[6] O. J. Dahl, E. W. Dijkstra, and C. A. Hoare. *Structured programming*. 1972.

[7] B. Du Bois, P. Van Gorp, A. Amsel, N. Van Eetvelde, H. Stenten, S. Demeyer, and T. Mens. A discussion of refactoring in research

and practice. *Reporte Técnico. Universidad de Antwerpen, Bélgica*, 2004.

[8] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools - CoSET*, 2000.

[9] P. Ebraert and Y. Vandewoude. Influence of type systems on dynamic software evolution. In *the electronic proceedings of the 21st International Conference on Software Maintenance - ICSM 2005*, 2005.

[10] M. Eid, M. Gollwitzer, and M. Schmitt. *Statistik und Forschungsmethoden: Lehrbuch*. Beltz Psychologie Verlags Union, 1st edition, 2010.

[11] B. Foote and R. E. Johnson. Reflective facilities in smalltalk-80. pages 327–335. ACM Press, 1989.

[12] M. Fowler. *Refactoring : improving the design of existing code*. Addison-Wesley, Reading MA, 1999.

[13] E. Gamma, R. Helm, and R. E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1st edition, Oct. 1994.

[14] A. Goldberg. *Smalltalk-80 : the language and its implementation*. Addison-Wesley, Reading Mass., 1983.

[15] I. Griffiths. *Programming C# 4.0.* Oreilly & Associates Inc, 2010.

[16] R. Johnson and W. Opdyke. Refactoring and aggregation. *Object Technologies for Advanced Software*, pages 264–278, 1993.

[17] Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin. Automated support for program refactoring using invariants. pages 736–743. IEEE Computer Society, 2001.

[18] M. Lehman. Laws of software evolution revisited. *Software process technology*, page 108–124, 1996.

[19] M. Lippert and S. Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 2006.

[20] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb. 2004.

[21] T. Mens, T. Tourwe, and F. Munoz. Beyond the refactoring browser: advanced tool support for software refactoring. In *The Proceedings of the 6th International Workshop on Principles of Software Evolution, IWPSE 2003.*, pages 39–44, Helsinki, Finland, 2003.

[22] T. Mens, A. Van Deursen, et al. Refactoring: Emerging trends and open problems. In *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects - REFACE. University of Waterloo*, 2003.

[23] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, Sept. 2008.

[24] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[25] D. Poo. *Object-oriented programming and Java*. Springer, London, 2nd edition, 2008.

[26] J. Rainsberger. Use your singletons wisely. *IBM developerWorks*, 2001.

[27] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A. M. Dery. Dynamic type inference to support object-oriented reengineering in smalltalk. *Lecture Notes in Computer Science*, 1543:76–77, 1998.

[28] D. Roberts and J. Brant. "Good enough" analysis for refactoring. In *European Conference on Object-Oriented Programming*, page 81–82, 1998.

[29] D. Roberts and J. Brant. Tools for making impossible changes – experiences with a tool for transforming large smalltalk programs. *IEE Proceedings - Software*, 151(2):49, 2004.

[30] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[31] S. Spoon and O. Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. *The 18th Conference on Object-Oriented Programming - ECOOP 2004*, page 485–493, 2004.

[32] F. Tip, R. M. Fuhrer, A. Kie¿un, M. D. Ernst, I. Balaban, and B. D. Sutter. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems*, 33:1–47, Apr. 2011.

[33] M. Unterholzner. Refactoring support for smalltalk using static type inference. Master's thesis, Free University of Bolzano, 2012.

[34] J. M. Utts and R. F. Heckard. *Mind on Statistics*. Duxbury, 4th edition, Jan. 2011.

[35] J. Van Gurp and J. Bosch. Design erosion: problems and causes. *Journal of systems and software*, 61(2):105–119, 2002.

[36] R. R. Wilcox. *Fundamentals of modern statistical methods substantially improving power and accuracy*. Springer, New York, 2010.

[37] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18:1038–1044, Dec. 1992.

[38] A. Wright. Type theory comes of age. *Communications of the ACM*, 53:16, Feb. 2010.

[39] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.

Tracking Down Software Changes Responsible for Performance Loss

Juan Pablo Sandoval Alcocer

Department of Computer Science (DCC)
University of Chile, Santiago, Chile

ABSTRACT

Continuous software change may inadvertently introduce a drop in performance at runtime. The longer the performance loss remains undiscovered, the harder it is to address. Current profilers do not efficiently support performance comparison across multiple software versions. As a consequence, identifying the cause of a slow execution caused by a software change is often carried out in an ad-hoc fashion.

We propose multidimensional profiling as a way to repeatedly profile a software execution by varying some variables of the execution context. Having explicit execution variation points is key to understanding precisely how a performance aspect evolves along with the version history of the software. We present the key ingredients to make multidimensional profiling effective, and sketch the design of Rizel, an implementation in the Pharo programming language.

1. INTRODUCTION

Measuring changes in the performance of an application is essentially realized by varying some parameters and profiling the program execution for each variation. Identifying which method is slower, for which argument and on which object, is crucial to precisely understanding the reason for a slow or fast execution. Moreover, an optimal execution is often used as a target for not-so-optimal executions. Caches are inserted and optimizations are implemented until the performance of a not-so-optimal execution is close enough to the optimal one.

This work is essentially conducted by software engineers in an ad-hoc manner. A set of benchmarks are manually constructed to measure the application performance for each slight variation. Typical variations include the size of the data input, a version of an algorithm or a particular sequence of function executions. As surprising as it may seem, current profilers are either unable to compare multiple executions or only offer superficial comparison facilities.

Before going into detail about the existing profilers, consider the following situation that was faced during the development of Mondrian[1], an agile visualization engine. Mondrian displays an arbitrary set of data as a graph in which each node and edge has a graphical representation shaped with metrics and properties computed from the data.

About two years ago, an optimization was implemented that made Mondrian 30% faster [3]. The optimization was carefully measured with a set of benchmarks. During the last two years, Mondrian has been in continuous development. And as it has gained new users, new requirements have been implemented to satisfy them. Whereas the range of offered features has grown, the performance of Mondrian has slowly decreased for some of the benchmarks. The optimization that made Mondrian 30% faster seems to have somehow vanished.

Tracking down the software changes that are responsible for this loss of performance is not easy, essentially because of the lack of adequate tools. Consider MessageTally the standard profiler of Pharo. It reports the CPU time consumption for each method for an application execution. The comparison of two profiles to identify the difference of execution has to be done manually, which is a tedious and laborious task in addition to be error prone. The commonly-used Java profilers [2]. Xprof[3] is built in the Java virtual machine and is essentially used by the Just-in-time compiler. Hprof[4] is the profiler promoted by Oracle. Both Xprof and Hprof does not provide comparison facilities. JProfile[5] and YourKit[6] are two popular commercial Java profilers. Both support a comparison of profiles by indicating the difference in absolute and relative CPU consumption time of each method. Although useful to keep track of the overall performance, knowing the difference between method execution times is often insufficient to understand the reasons for the performance variation. In addition, a profiled call graph may significantly differ from two profile reports, which seriously complicates the analysis.

To understand the reasons for a slow execution caused by software evolution, we might ask:

- *How can we reproduce the performance degradation?*
- *How can we identify the piece of code that is the responsible for the loss of performance?*

[1]`http://moosetechnology.org/tools/mondrian`
[2]We have conducted all our experiments in the Pharo programming language.
[3]`http://bit.ly/xprofiler`
[4]`http://bit.ly/hprofiler`
[5]`http://www.ej-technologies.com`
[6]`http://www.yourkit.com`

When applied to our example with Mondrian, Xprof, Hprof, JProfiler and YourKit are useless at answering any of these questions. The reason is that the profile comparison exercised by JProfiler and YourKit does not capture all the variables that these questions refer to, such as the benchmarks and software versions. Being able to profile an application along several variables is the topic of our work.

In this article we propose *multidimensional profiling* as a way to repeatedly profile a software execution. We present the key ingredients to make multidimensional profiling effective. We accomplish this by changing the values of some variables of the execution context. Having explicit execution variation points is key to precisely understand how the performance of a particular feature evolves along the version history of the software.

We present *Rizel*, a multidimensional profiler that considers two dimensions: benchmarks and software versions. *Rizel* provides two visualizations where the performance degradation clearly and explicitly appears: The *Performance Comparison Matrix* shows the performance of a software in difference versions, allowing us to find a breakpoint in performance. The *Performance Evolution Blueprint* allows one to understand the performance impact of changing the definition of a method. Finally, we employ *Rizel* to identify which software changes are responsible for performance loss in a real world application, XMLSupport.

This paper is structured as follows. Section 2 introduces Multidimensional Profiling. Section 3 presents Rizel, our multidimensional profiler. Section 4 describes the notion of filtering, in order to reduce the amount of data displayed. Section 5 employs Rizel to identify software changes responsible for performance loss, in XMLSupport. Section 7 briefly presents the related work. Section 6 discusses the experience we gained. Section 8 concludes and sketches our future work.

2. MULTIDIMENSIONAL PROFILING

We define *multidimensional profiling* as the activity of reasoning about a software execution by varying multiple variables related to its execution. These variables could be function inputs, benchmarks or software versions, among others. Our objective is to gain a better understanding of a software execution by relating different profiles obtained from slightly different conditions. Opportunities to optimize and to minimize resource consumption are then easier to find.

The rationale behind multidimensional profiling is that if a software execution is particularly fast or slow for an identified situation (*i.e.,* particular values for the variables), then the situation can be exploited to improve the overall execution.

In a nutshell.

The ingredients to accurately exercise multidimensional profiling are:

- *Definitions of the variation points of the executing environment.* The variation points are defined to be a set of variables $(V_1, ..., V_n)$. Each of these variables is associated with a particular aspect of the execution environment, such as a software version, benchmark, parameters of a particular method, instances of a particular class.

- *Specification the values of each variation point.* Each variable may either be set to a fixed value, or may iterate over a range of values. Each value produces a new profile. To better measure the impact of a variable evolution, it is preferable to have all but one variable fixed. These executions result in a set of profiles $P_1, ...P_m$.

- *Stable profiles.* Each execution has to be repeatable and isolated from other executions. This means that two profiles P_j and P'_j produced by two identical executions have to be "close enough" to be meaningful.

- *Presentation of the results.* Data must be presented in such a way that variation performance clearly and explicitly appears. The evolution of V_i has to be unambiguously represented to be able to draw a conclusion about the performance evolution that results.

3. IMPLEMENTATION

We have prototyped Rizel [7], a multidimensional profiler. Rizel is implemented in the Pharo Smalltalk language, and is based on Spy [4], a flexible and open instrumentation-based profiler framework. The set of variables that Rizel currently considers are benchmarks and software versions. This means that for a given software, Rizel can:

- run a particular benchmark b for each of the software versions $s_1, ..., s_k$

- run a different benchmark $b_1, ..., b_l$ for a particular software version s

The following script uses Rizel to measure the performance of Mondrian for four benchmarks over nine representative versions.

```
1  rizel define: #input
2    named: #version
3    with: self mondrianVersions.
4  rizel define: #input
5    named: #benchmark
6    with: #( #treeLayout #manyNodes #openSimpleGraph
        #subviews).
7
8  results := rizel execute: [:version :benchmark |
9      version load.
10     rizel profile: [
11       XMLSupportBenchmarks new perform: benchmark ]
12       inPackagesMatching: 'XML−∗']].
13
14 rizel display: results.
```

Lines 1-6 define benchmarks and software versions as variation points of the executing environment. Lines 8-12 define the code that should be executed for each variation. These executions results in a set of profiles.

Our profiler measures the number of sent messages by each method. Bergel has shown [2] that counting message accurately estimates the execution time without the inconveniences usually associated to time measurement and execution sampling. For example, counting messages is significantly more stable than directly measuring the time: profiling the same execution twice results in two very close profiles.

Rizel presents the result with two visualizations: *Performance Comparison Matrix*, to detect which version introduces the performance degradation and for which benchmark,

[7] `http://users.dcc.uchile.cl/~jsandova/rizel/`

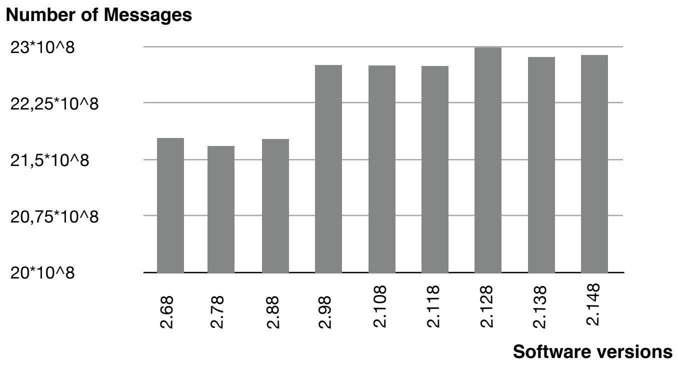


Figure 1: Performance degradation of the #openSimpleGraph benchmark of Mondrian

and *Performance Evolution blueprint*, to identify methods that change the source code and understand the impact of those changes in the performance.

Performance Comparison Matrix.

The Performance Comparison Matrix shows the performance evolution of a number of benchmarks across software revisions. Figure 2 shows the evolution of the benchmarks against the versions of Mondrian.

Each row of the matrix corresponds to a single benchmark performance evolution. A color is assigned to each cell in the row; white is associated to the lowest execution time and black to the highest execution time across all versions. For example, the third row correspond to the benchmark of `#openSimpleGraph` across nine Mondrian versions (Figure 1), where the version 2.78 has the lowest execution time (white box) and the version 2.128 has the highest execution time (black box).

We see that each benchmark indicates a progressive degradation of the performance of Mondrian. Each of these benchmarks corresponds to a particular feature. Each feature is getting slower, not at the same pace, *e.g.,* #openSimpleGraph and #treeLayout are consuming much more time after Version 2.98. Execution time of #subviews increases after Version 2.88.

Performance Evolution blueprint.

The Performance evolution blueprint helps compare two profiles in order to identify the piece of code that introduces the loss of performance. It is graphically rendered as a polymetric view [5] an example is show in Figure 3. A polymetric view is a lightweight software visualization enriched with software metrics. We consider that two profiles capture the performance degradation. When the profiles correspond to different software versions, and the old version profile is faster than the new version profile.

Nodes represent methods and edges represent method invocations (upper methods invoke lower ones). Each node in the call graph has the following associated metrics:

- **Width** is proportional to the difference between number of executions of the newest version and older version in logarithmic scale. If the difference is negative, the width is five pixels as default. If the method node is wider, it means the method is executed more times than previous version.
- **Height** is proportional to the difference between the number of sent messages of the newest version and older version in logarithmic scale. Only positive values are meaningfully represented. The larger the height, the slower the new method version. In case the new version is faster, the height has a minimum of five pixels. The motivation behind this is to favor the identification of slowdown.
- **Node Color** is assigned with the following criteria: *Green*: method sends less messages before (*i.e.,* the new method version is faster); *Light Red*: method sends more messages than before, and it is executed the same amount of times as before; *Red*: method sends more messages than before, and it is executed more than before; *Yellow*: method was not implemented in previous version; *White*: method number of sent messages is identical.
- **Border Color** is assigned with the following criteria: *Red*: method definition has been redefined (the method source code has changed between the two version); *Black*: the method source code has not changed.

For example, Figure 3 shows the call graph that compares profile [#treeLayout ; 1.38] and profile [#treeLayout ; 1.48] of Mondrian. The complete call graph involves 2 246 nodes. Scalability is therefore an issue. Figure 3 shows the partial call graph that highlights differences. In the following section we illustrate how this graph was reduced.

Of the thirteen changed methods (boxes with red border) shown in Figure 3 only two impact the performance: *MOAnnouncer >>popupText:delay:* changed and the new version of it sends more messages than in its previous version; this change impacts the unchanged *MOAnnouncer >>popupView:delay:*, but it sent more messages than before too. And *MOAnnouncer>> popupView:delay:zoomedInBy:* changed and sent more messages and is executed more times than in the previous version.

The other eleven boxes with a red border are modified methods that do not impact the performances: their number of sent messages and their number of executions are identical than previous version that is why they are small and do not have any relationship.

4. REDUCING CALL GRAPH

A common problem in visualizing runtime information is the scalability, since a simple execution can generate a vast amount of information. For example, a simple execution of a Mondrian benchmark executes about two thousand different methods. However, we are only interested in the methods whose the source code has changed and the impact of this change in the other methods.

Consider the call graph in Figure 4. The method m4 is not important for understanding the reason of a slow execution, because it has the same number of messages and the number of executions as before. This means that the change does not impact this method. The method m2 is not important for two reasons: (1) it has no relation with the method m3 that change in the of source code. (2) m2, like the method m4, has the same number of messages as before.

Method m1 is not important either, even if it sends more messages than before. Because, to calculate the number of sent messages of m1, we consider all classes and methods

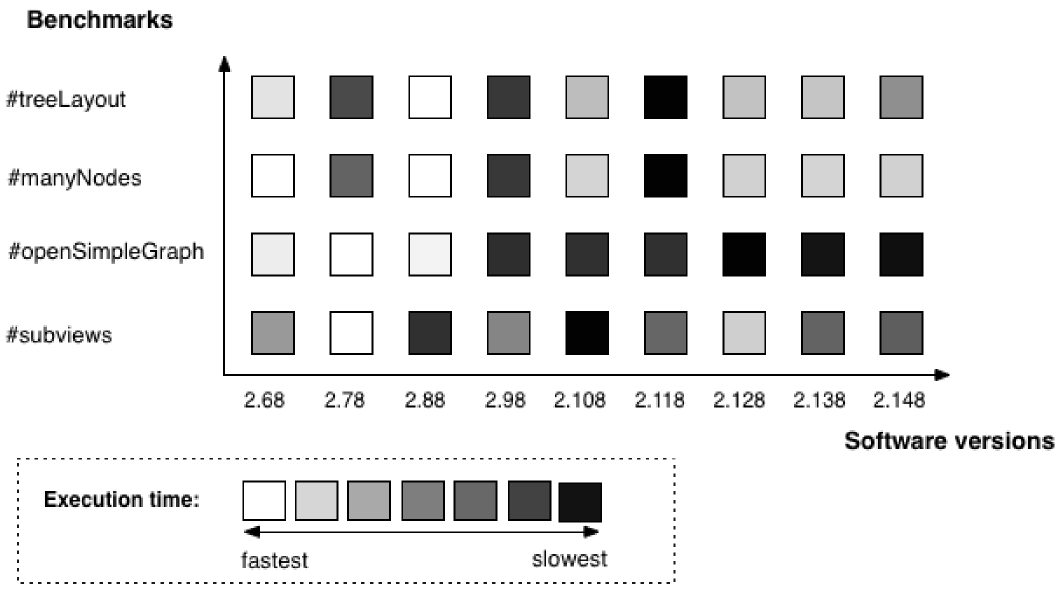


Figure 2: The *Performance Comparison Matrix* shows the performance evolution of 4 benchmarks over 9 versions of Mondrian. We can see the performance of some of the benchmarks that have slowly decreased over the different versions.

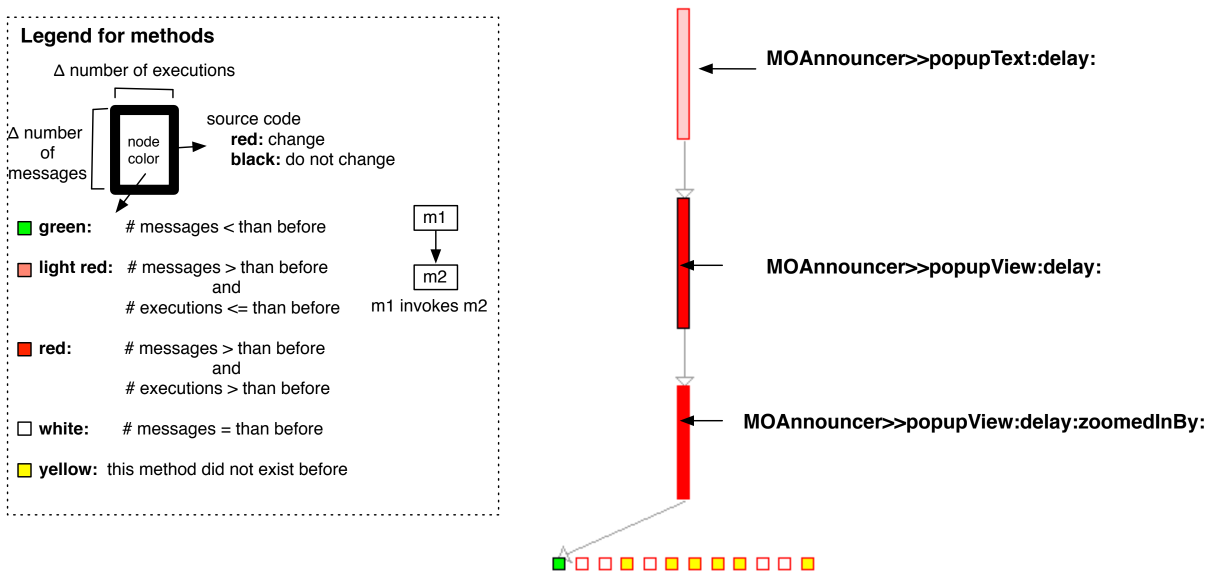

Legend for methods
Δ number of executions
Δ number of messages
node color
source code
red: change
black: do not change
green: # messages < than before
light red: # messages > than before and # executions <= than before
red: # messages > than before and # executions > than before
white: # messages = than before
yellow: this method did not exist before
m1
m2
m1 invokes m2
MOAnnouncer>>popupText:delay:
MOAnnouncer>>popupView:delay:
MOAnnouncer>>popupView:delay:zoomedInBy:

Figure 3: The *Performance Evolution Blueprint* compares the profiles [#treeLayout ; 2.138] and [#treeLayout ; 2.148]; showing the methods that have had source code changes (boxes with red border) and the impact of such changes in the performance (red and large boxes).

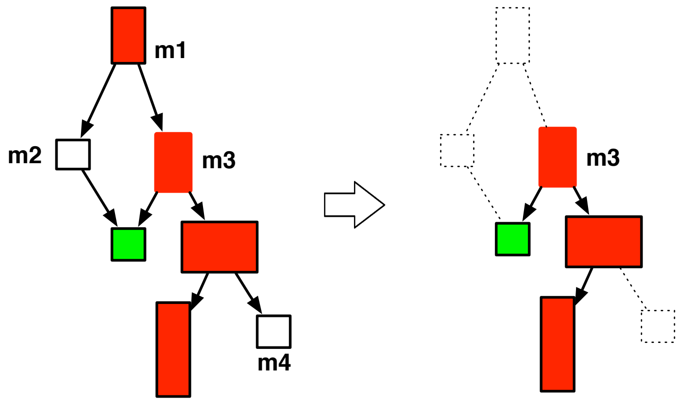


Figure 4: Hiding unnecessary methods to understand why we have a slower execution, in order to reduce the call graph.

involved in the execution, including calls made by m2 and m3. And the only reason that m1 sent more messages is because m3 sent more messages than before.

We define two simple rules to determine if a method is relevant to understand the reason for a slow execution or not, to reduce the call graph.

- A method is relevant if it's source code has changed.
- A method is relevant if it sends different number of messages than previous version and it was invoked by an relevant method.

Note that the second rule excludes methods that have a different number of executions, because if a method has a different number of executions it should send a different number of messages.

5. CASE STUDY

We used *Rizel* to analyze *XMLSupport*, a library for packages that parse, manipulate and generate XML documents in Pharo smalltalk.

We write a few lines of Pharo code to exercise multidimensional profiling with *Rizel*. We need to review only the five last software versions to find a performance degradation in *XMLSupport*.

Figure 5 shows that all benchmarks have a performance degradation in version 1.1.7. At this point we know how to reproduce the performance failure. And we only need to compare a profile from 1.1.6 and 1.1.7 versions of any benchmark. To confirm this, we use MessageTally, the sampling profiler for Pharo smalltalk to calculate the execution time of the benchmark #xmlDomParser for the version 1.1.6 and 1.1.7, changing the version manually and getting the same results as Rizel. There is a variation of 10% in the execution time between versions.

Figure 6 shows the *performance evolution blueprint* that compare *profile [#xmlDomParser ; 1.1.6]* and *profile [#xmlDomParser ; 1.1.7]*.

Consider the particular case of the method *XMLTokenizer >>nextName* that sends many more messages than before. Furthermore, this method is responsible for the fact that *XMLNestedStreamReader>>atEnd* is executed more times and is sending more messages than before. The opposite

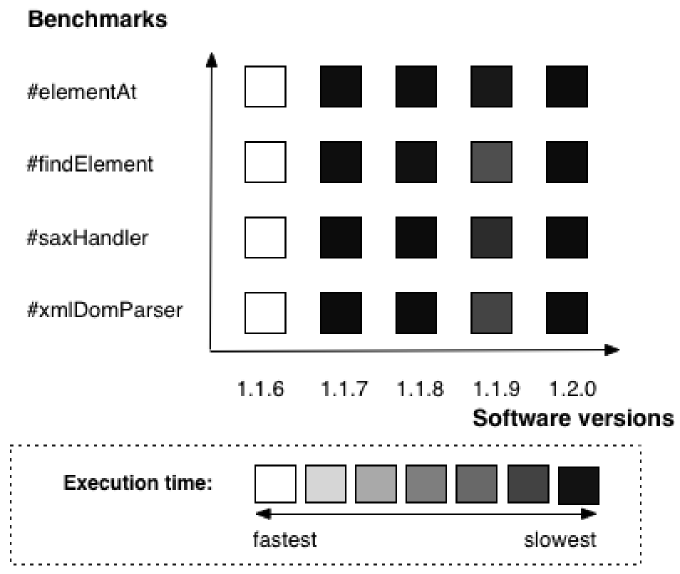


Figure 5: The *Performance Comparison Matrix* shows that all benchmarks in XMLSupport have had performance degradation since version 1.1.7.

happens with the other methods shown with red border. These have no impact in their outgoing methods.

The methods with white boxes whose code changed but have the same number of sent messages and executions as before. The yellow boxes are methods that do not exist in the previous version. All of these methods have no impact on their outgoing methods like *XMLTokenizer >>nextName*. The green methods have less sent messages than before.

We found that in the version 1.1.6, XML names are matched heuristically; the tokenizer would read characters until it encountered one of a few that it knew could not rightfully be part of an XML name.

```
XMLTokenizer>>nextName
  | nextChar |
  ^ streamWriter writeWith: [:writeStream |
     [(nextChar := streamReader peek) isNil
        or: [NameDelimiters includes: nextChar]]
           whileFalse: [writeStream nextPut: streamReader
     next].
     writeStream position > 0
        ifFalse: [self errorExpected: 'name'].
     writeStream stringContents]
```

And in the version 1.1.7 the the method *XMLTokenizer >>nextName* was changed. The XML name properly matches the entire range for the first character of a name and for subsequent characters, as specified by the XML specification.

```
XMLTokenizer>>nextName
  | nextChar |
  ^ streamWriter writeWith: [:writeStream |
     (NameStartChars includes: (nextChar :=
     streamReader next))
        ifFalse: [self errorExpected: 'name'].
     writeStream nextPut: nextChar.
     [streamReader atEnd not
```

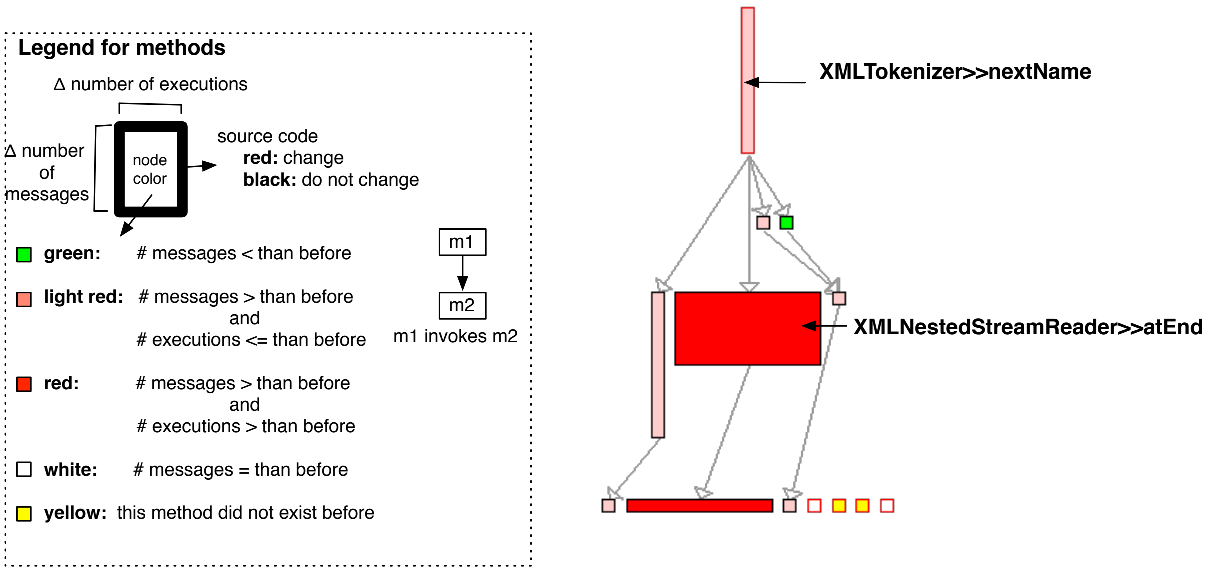

Legend for methods
Δ number of executions
Δ number of messages
node color
source code
red: change
black: do not change
green: # messages < than before
light red: # messages > than before and # executions <= than before
red: # messages > than before and # executions > than before
white: # messages = than before
yellow: this method did not exist before
m1
m2
m1 invokes m2
XMLTokenizer>>nextName
XMLNestedStreamReader>>atEnd

Figure 6: The *Performance Evolution Blueprint* comparing profiles [#xmlDomParser ; 1.1.6] and [#xml-DomParser ; 1.1.7].

```
        and: [(NameStartChars includes: (nextChar :=
    streamReader peek))
            or: [AdditionalNameChars includes: nextChar]]]
        whileTrue: [writeStream nextPut: streamReader next].

    writeStream stringContents]
```

6. DISCUSSION

We discuss a number of points about *Rizel*.

Assumptions – There are a number of assumptions behind *Rizel*. First, we know in advance that our application is slower than before. Second, we know that software versions are stable and the benchmarks that represents features of software can be executed in most of the versions, without throwing an exception.

Scalability – In the Section 4 we proposed a way to filter the call graph. We experiment with small and medium sized applications. Comparing nearby versions that have few changes between versions, small and intuitive visualizations. We have no evidence if our approach does not scale since we have not a Pharo application that is sufficiently large to produce an excessive visualization found.

Using *Rizel* after each commit can be a good way to address these points, and could have a number of advantages. For example, a performance failure could be detected earlier and the developer only has to compare the new version with the previous one.

7. RELATED WORK

Comparing program elements between two executions provides a means for developers to better understand a program's performance variation. There are a number of techniques for dynamically comparing two program executions.

Zhuang *et al.* [8] propose *PerfDiff* a framework for analyzing performance across multiple runs of a program, possibly in a dramatically different execution environment. Their framework is based on a lightweight instrumentation technique for building a calling context tree (CCT) of methods at runtime. Mostafa and Krints [7] present *PARCS*, an offline analysis tool that automatically identifies differences between the execution behavior of two revisions of an application. *PARCS* collects program behavior and performance characteristics via profiling and generation of calling context trees. Both comparing CCTs identifying performance-attribute (e.g execution time, invocation count) and topological differences (e.g. added, deleted, modified, and renamed methods). Likewise, we compare the number of messages and number of executions using a Call Graph, also detecting modified and new methods.

Adamoli *et al.* present Trevis, an extensible framework for visualizing, comparing, clustering, and intersecting CCTs [1]. To scale their tree visualization, they use calling context tree ring charts (CCRC) and just reduce the thickness of a ring segment, which leads to a reduction of the diameter of the visualization [6]. In our approach we use a call graph and provide a simple mechanism to display only the important methods for the analysis.

Performance Evolution blueprint is based on a behavioral evolution blueprint presented by Bergel [3]. The main difference with our blueprint is that we use different metrics, for example, we use the number of sent messages to estimate the execution time in order to get replicable results. Another difference is that in behavioral evolution blueprint the width is determined by the number of executions. In our case, the width is the difference of the number of executions in order to know if a method is executed more times than the previous versions. This helps us to know whether or not a method spends more time because it is executed more times than before.

Our approach compares multiple profiles obtained even under slightly different conditions, counting messages to estimate the execution time and comparing the profiles. It allows one to determine which versions, and in which benchmark the performance failure should be executed to reproduce. And finally, it compares two executions, visualizing metrics like the number of sent messages and number of executions to detect and understand the possible cause of a drop in performance.

8. CONCLUSION AND FUTURE WORK

Multidimensional profiling is an innovative approach to measure software performance: crystalizing the performance of each software feature into a set of dedicated benchmarks makes it possible to precisely monitor the global performance of a software against different versions.

We present Rizel, a multidimensional profiler that considers two dimensions: benchmarks and software versions. Rizel provides visualizations where the performance degradation clearly and explicitly appears. We use Rizel to analyze two Pharo applications, to get to the root cause of a slowdown by identifying the method revision responsible for slower performance.

We will then concentrate on identifying patterns to describe the evolution of feature performance across multiple software versions. As far as we are aware, all of these points have not been researched extensively by the research community on software performance.

9. REFERENCES

[1] Andrea Adamoli and Matthias Hauswirth. Trevis: a context tree visualization & analysis framework and its use for classifying performance failure reports. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 73–82, New York, NY, USA, 2010. ACM.

[2] Alexandre Bergel. Counting messages as a proxy for average execution time in pharo. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11)*, LNCS, pages 533–557. Springer-Verlag, July 2011.

[3] Alexandre Bergel, Felipe Bañados, Romain Robbes, and Walter Binder. Execution profiling blueprints. *Software: Practice and Experience*, August 2011.

[4] Alexandre Bergel, Felipe Bañados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. *Journal of Computer Languages, Systems and Structures*, 38(1), December 2011.

[5] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[6] Philippe Moret, Walter Binder, Alex Villazón, and Danilo Ansaloni. Exploring large profiles with calling context ring charts. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 63–68, New York, NY, USA, 2010. ACM.

[7] Nagy Mostafa and Chandra Krintz. Tracking performance across software revisions. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 162–171, New York, NY, USA, 2009. ACM.

[8] Xiaotong Zhuang, Suhyun Kim, Mauri io Serrano, and Jong-Deok Choi. Perfdiff: a framework for performance difference analysis in a virtual machine environment. In *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 4–13, New York, NY, USA, 2008. ACM.

Spec

A Framework for the Specification and Reuse of UIs and their Models

Benjamin Van Ryseghem
RMoD, Inria Lille – Nord Europe
benjamin.van_ryseghem@inria.fr

Stéphane Ducasse
RMoD, Inria Lille – Nord Europe
stephane.ducasse@inria.fr

Johan Fabry
PLEIAD Lab – Computer Science Department (DCC)
University of Chile
jfabry@dcc.uchile.cl

Abstract

Implementing UIs is often a tedious task. To address this, UI Builders have been proposed to support the description of widgets, their location, and their logic. A missing aspect of UI Builders is however the ability to reuse and compose widget logic. In our experience, this leads to a significant amount of duplication in UI code. To address this issue, we built Spec: a UIBuilder for Pharo with a focus on reuse. With Spec, widget properties are defined declaratively and attached to specific classes known as composable classes. A composable class defines its own widget description as well as the model-widget bridge and widget interaction logic. This paper presents Spec, showing how it enables seamless reuse of widgets and how these can be customized. After presenting Spec and its implementation, we discuss how its use in Pharo 2.0 has cut in half the amount of lines of code of six of its tools, mostly through reuse. This shows that Spec meets its goals of allowing reuse and composition of widget logic.

1. Introduction

Building user interfaces is a notoriously time-consuming task. To help developers in their tasks, several approaches have been proposed previously. The basic principle of decoupling the model from its view: MVC [5] was first proposed in Smalltalk-80. This principle was later evolved to Model View Presenter (MVP) [10] by the Taligent project. In MVP, the presenter assumes part of the functionality of the controller and is the sole responsible for coordinating how the UI manipulates the underlying model. The view is now also responsible for handling UI events, which used to be the controller's responsibility.

Orthogonally to these concepts, UI builders were developed as tools to facilitate the design and building of UIs. The goals of UI builders are often twofold: firstly to support the description of widgets (location, size, color) and their logic, and secondly the composition and reuse of existing component logic. VisualWorks [4, 9] was a pioneer of this approach. Its builder is based on application classes that glue widgets and domain objects together, based on a literal description of widget properties.

Important issues with UI builders stem from the fact that their working is often based on direct code generation from the UIBuilder visual pane. As a first consequence the simple fact of reloading a UI description in the builder for editing, arguably a common occurrence, is already a complicated process. This complication arises because the UI description code has to be interpreted differently from a normal execution, since the normal execution opens the UI for use. Secondly, there is still the challenge of reusing widgets and their interaction logic. In our experience with Pharo, the use of UI builders there has led to a significant amount of code duplication which can be avoided. For example, the Senders/Implementors tool shows a list of methods and displays their source code. Pharo also provides the VersionBrowser, which displays methods as a list and their source code. Furthermore the ProtocolBrowser displays all the methods of a class and their source code. These three tools are mostly duplicated code, essentially implementing three times the same behavior and the same UI, with some superficial differences. In our opinion this code duplication arises because the widgets are not generic enough to be reused while also being able to be adapted to cope with (subtle) variations.

To address the above issues, two underlying design choices need to be taken: (1) how do we define UI descriptions and (2) how do we compose the logic of UIs. We assert that there is a need for a declarative way to specify UIs that also allows for seamless composition and reuse of the UI declaration and logic. In line with this assertion we have de-

IWST '12 August 28th, 2012, Gent, Belgium.
Copyright © 2012 ACM [to be supplied]. . . $10.00

veloped Spec, a UI builder for Pharo, and we present it in this article.

With Spec, widget properties are defined declaratively and attached to specific classes known as composable classes. These composable classes also act as presenters by defining the bridge to the underlying model, in addition to the widget interaction logic. Spec reuse includes these presenters, i.e. Spec allows for the reuse of widget logic and their composition as well as their connection to the underlying model. This support for reuse is not only novel but we also consider it the most important contribution of Spec, as its use removed a high amount of code duplication in Pharo 2.0.

This paper is structured as follows: we next detail the issues that emerge from the current UI builder approach. This is followed, in Section 3, by an introduction of Spec that builds a number of UIs highlighting reuse. Section 4 gives an overview of the more salient points of its implementation and, in Section 5, we provide a more formal description of the different elements used in the example. Spec is currently used in Pharo, and we talk about this in Section 6. Related work is discussed in Section 7, and Section 8 concludes.

To avoid any ambiguities in this text due to issues with terminology, we first define three terms briefly, since these typically have overloaded meanings.

UI Element: an interactive graphical element displayed as part of the Graphical User Interface.

UI Model: an object that contains the state and behavior of one or several UI elements.

Widget: the union of a UI Element and its UI model.

2. UI Builder Challenges

A UI builder is a tool used to generate user interfaces. Such builders help the developers by providing a framework for UI construction on top of the UI libraries provided by the language. They may also provide a UI for graphically building a UI. Put differently, a UI builder is *not necessarily* a tool with a UI although it may provide a UI to interactively place widgets on a canvas.

To be able to help the developers in creating UIs, UI builders usually provide support for:

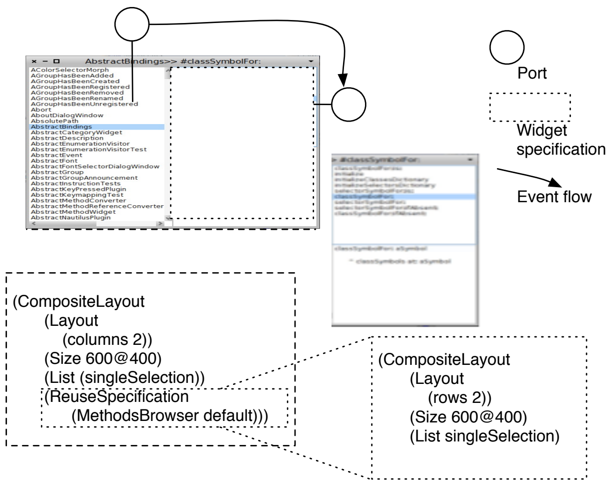
- the description of widgets (color, size, visual effect, specific behavior,...)
- the description of their placement
- the definition of the widget behavior, *e.g.,* how it reacts to certain events.

However in our experience the above is not enough. This is because developing UIs is not only about widget generation, but also about the reuse of the logic between widgets. As Pharo maintainers we have seen that most of the UIs of the tools present in Pharo were written from scratch. This even if a lot of tools are essentially manipulating the same objects and rendering them more or less the same way. This lack of reuse makes the system harder to maintain and slows down enhancements to the UIs of these tools. To address these problems, UI builders should also support the reuse of widget logic and composition. We have seen that the logic of one widget is often based on the wiring of the logic of adjacent or nested widgets. Hence being able to compose and reuse existing behavior is central to be able to build new widgets.

The goal of reusability however brings a new problem. Indeed, if the widgets must be reusable it means that on one hand the widgets must be generic enough to be used in different scenarios and on the other hand they should be parametrizable enough to fit these new scenarios.

To enable the reuse in the process of building and maintaining the UIs of Pharo, we have built Spec. Spec is a new UI builder whose goal is to support UI interface building and seamless reuse of widgets. Spec is based on two core ideas: first the declarative specification of the visual behavior of widgets and their logic, and second inherent composability of widgets, based on explicit variation points.

Figure 1 shows the principles of Spec: a UI is built from composed widgets that are glued together using ports and whose visual characteristics are defined using a declarative specification that are reused over composition.



```
(CompositeLayout
    (Layout
        (columns 2))
    (Size 600@400)
    (List (singleSelection))
    (ReuseSpecification
        (MethodsBrowser default)))
```

```
(CompositeLayout
    (Layout
        (rows 2))
    (Size 600@400)
    (List singleSelection)
```

Figure 1. Spec principles

3. Spec by Example

In this section we introduce Spec and its key points by showing an example of the typical use of Spec. Note that we only focus here on the logic of the building process, the discussion of a graphical tool to compose widgets based on Spec is out of the scope of the paper.

The example we develop in this section starts with showing how basic widgets are composed to build a simple UI and continues with illustrating how these composed widgets can be reused and adapted to build more substantial UIs. In total we build three UIs: a Method List that shows a collection of methods, in Section 3.1, a Method Browser that reuses the list and adds a pane showing the source code of the method, in Section 3.2, and lastly a Class Browser, reusing and adapting the Method Browser, in Section 3.3.

3.1 Methods List

In this section, we present how to build a method list in five steps:

1. the creation of the class;
2. the implementation of the initialize process;
3. the implementation of the getters;
4. the specification of a layout;
5. the window title.

We will now present these five steps in more detail.

Class Creation. First we need to create a class named MethodsList.

```
ComposableModel subclass: #MethodsList
  instanceVariableNames: 'list'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'IWST12-Spec'
```

Here we can see two things:

- the superclass is ComposableModel: this class is the root of the Spec UI model hierarchy.
- the instance variable *list*: an instance variable needs to be defined for holding the UI model that will represent our methods list.

Initialization. Second, the initialization. The initialization of a UI is done in three different methods:

1. initializeWidgets: to set the instance variables which hold sub models and their associated widgets and to configure these sub UI models;
2. initializePresenter: to wire sub UI models together;
3. initialize: to initialize remaining state of the UI.

In this example, only the widget instantiation has to be done in the initializeModels method of the MethodsList class, such that it sets the *list* instance variable to contain a ListComposableModel with its associated list UI element.

```
MethodsList>>initializeWidgets
   self instantiateModels: { #list -> #ListComposableModel }.
```

The code above shows how the model for the list is instantiated, in a declarative fashion. The method instantiateModels: allows one to provide a collection of associations where the key is the instance variable name and the value is a UI model class. Hence the code above creates a new instance of ListComposableModel (and its associated list UI element) and stores it in the instance variable *list*.

Accessors. Third the accessor to the instance variable has to be implemented, such that the Spec infrastructure can obtain this list instance when required.

```
MethodsList>>getList
   ^ list
```

Layout Specification. Fourth we specify a layout: the object that is used to describe and represent the layout of the UI elements. This is done by implementing a method that returns it on the class side of the UI model. For our example, we implement MethodsList class»myFirstSpec on the class side of MethodsList as follows:

```
MethodsList class>>myFirstSpec
   <spec: #default>

   ^ SpecLayout composed
         add: #getList;
         yourself.
```

Since multiple layouts per UI model can be present, there is a mechanism to set the layout to use by default. There are two ways to define the default specification to be used:

1. pragma: all specifications are tagged with a pragma <spec:> allowing the spec infrastructure to correctly retrieve the corresponding method. In addition, the keyword default can be used in the pragma to specify that this layout has to be used by default.
2. method name: if there is no <spec: #default> tag, the method named *defaultSpec* is used.

The code above uses a pragma and simply returns the layout object for this UI. Sending the message composed to the SpecLayout class yields a composed layout, which allows one to compose the different models that are part of Spec. To this composed layout, the add: message is sent, adding the argument to the layout. In this case we provide the symbol getList, the selector of the getter of the model we want to include into the methods list. Note that, in general, this argument may be a SpecLayout as well, allowing for the reuse of high level composed models, as we shall show later.

Executing the following snippet displays the generated widget embedded in a window using the above, default, specification.

```
MethodsList new openWithSpec.
```

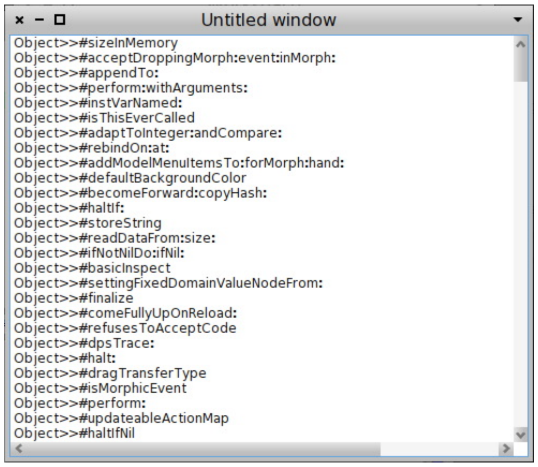
To populate the list, the message items: has to be sent to the instance variable list with a collection of items to be shown as argument. In our example this would be a collection of methods, *e.g.,* the methods of the class Object as shown below:

```
(MethodsList new
   openWithSpec;
   yourself)
   getList items: Object methods
```

By default, the method printString is sent to each item to produce the string used to display the item in the list. Often the default behavior displays to much information, or is not accurate enough. To address this, a block can be used to specify how to generate the display string. This is achieved by sending the message displayBlock: to the list widget. The following code provides an example of how to specify a display block in the initializeWidgets method such that the displayed string follows the form class name»selector.

```
MethodsList>>initializeWidgets
    self instantiateModels: { #list -> #ListComposableModel }.
    list displayBlock: [:method || name |
        name := method methodClass name.
        name, ’>>#’, method selector ].
```

When the window is opened, each item is displayed as shown in Figure 2.



Untitled window

```
Object>>#sizeInMemory
Object>>#acceptDroppingMorph:event:inMorph:
Object>>#appendTo:
Object>>#perform:withArguments:
Object>>#instVarNamed:
Object>>#isThisEverCalled
Object>>#adaptToInteger:andCompare:
Object>>#rebindOn:at:
Object>>#addModelMenuItemsTo:forMorph:hand:
Object>>#defaultBackgroundColor
Object>>#becomeForward:copyHash:
Object>>#haltIf:
Object>>#storeString
Object>>#readDataFrom:size:
Object>>#ifNotNilDo:ifNil:
Object>>#basicInspect
Object>>#settingFixedDomainValueNodeFrom:
Object>>#finalize
Object>>#comeFullyUpOnReload:
Object>>#refusesToAcceptCode
Object>>#dpsTrace:
Object>>#halt:
Object>>#dragTransferType
Object>>#isMorphicEvent
Object>>#perform:
Object>>#updateableActionMap
Object>>#haltIfNil
```

Figure 2. The methods list with a specific display

Window Title. Fifth and last, we show how to change the window title according to the current list selection. By convention, the window title is determined by the return value of the method named title.

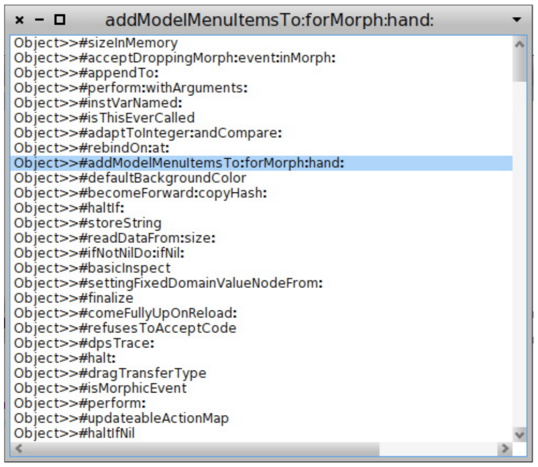
```
MethodsList>>title
    ^ list selectedItem
        ifNil: [ ’Methods list’ ]
        ifNotNil: [:method | method selector ].
```

The code above returns the selector of the selected method, or the string ’Methods List’ if no item is selected.

In order to update the title each time the selected item of the list has changed, we must relate the selection action in the methods list to the updating of the window title. This hence needs to be implemented in the initializePresenter method, and is as follows:

```
MethodsList>>initializePresenter
    list whenSelectedItemChanged: [ self updateTitle ].
```

The code of the method states that the title of the window should be updated when the selected item in the list changes. In Figure 3 we show the result of these changes: a window where the title is the selector of the selected item.



addModelMenuItemsTo:forMorph:hand:

```
Object>>#sizeInMemory
Object>>#acceptDroppingMorph:event:inMorph:
Object>>#appendTo:
Object>>#perform:withArguments:
Object>>#instVarNamed:
Object>>#isThisEverCalled
Object>>#adaptToInteger:andCompare:
Object>>#rebindOn:at:
Object>>#addModelMenuItemsTo:forMorph:hand:
Object>>#defaultBackgroundColor
Object>>#becomeForward:copyHash:
Object>>#haltIf:
Object>>#storeString
Object>>#readDataFrom:size:
Object>>#ifNotNilDo:ifNil:
Object>>#basicInspect
Object>>#settingFixedDomainValueNodeFrom:
Object>>#finalize
Object>>#comeFullyUpOnReload:
Object>>#refusesToAcceptCode
Object>>#dpsTrace:
Object>>#halt:
Object>>#dragTransferType
Object>>#isMorphicEvent
Object>>#perform:
Object>>#updateableActionMap
Object>>#haltIfNil
```

Figure 3. The methods list with a dynamic title

Public API. This completes the construction of the UI model. However if we want this UI model to be reused and embedded we must provide a public API for it. In our example, we want to have the API of MethodsList polymorphic to the core protocol of the embedded list. Hence for each method of the API we forward the message to the *list* instance variable, as shown below:

```
MethodsList>>items: aCollection
    list items: aCollection

MethodsList>>displayBlock: aBlock
    list displayBlock: aBlock

MethodsList>>whenSelectedItemChanged: aBlock
    list whenSelectedItemChanged: aBlock

MethodsList>>resetSelection
    list resetSelection

MethodsList>>selectedItem
    list selectedItem
```

Those methods will be used in the following section when defining its public API, which is used in our last example (in Section 3.3).

3.2 Methods Browser

To show how Spec allows for the reuse of existing models, the next step of our example builds a message browser which reuses the method list we just constructed. We will follow the same five steps as previously:

1. the creation of the class;

2. the implementation of the initialize method;
3. the implementation of the getters;
4. the specification of a layout;
5. the window title.

These steps will now be presented in more details.

Class Creation. First we define a class, this time named MethodsBrowser.

```
ComposableModel subclass: #MethodsBrowser
    instanceVariableNames: 'methodsList text'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'IWST12-Spec'
```

The class has two instance variables:

- *methodsList*: the list for displaying the methods. It will be an instance of the MethodsList class we defined in Section 3.1, i.e., we reuse the methods list we defined previously.
- *text*: the text zone used to display the source code. It will be an instance of TextModel.

Initialization. Second, in the initialize process, we have to instantiate the value for each instance variable.

```
MethodsBrowser>>initializeWidgets
    self instantiateModels:
      {  #methodsList     -> #MethodsList.
         #text            -> #TextModel }
```

The above code shows that a model that is being instantiated can be a standard class of Spec as well as any UI model class that has been defined using Spec. This is the key feature that allows for seamless reuse of models in Spec.

Next, we specify the overall behavior of the UI, linking the two widgets together. When an item from the list is selected, the text area should display its source code.

```
MethodsBrowser>>initializePresenter
    methodsList whenSelectedItemChanged: [:method |
        text text: (method
                    ifNil: [ '' ]
                    ifNotNil: [ method sourceCode ]) ]
```

The initializePresenter method above specifies the overall behavior of the UI. It links the list to the text field by stating that when a item in the list is selected the text of the text zone is set to:

- an empty string if no item is selected
- the source code of the selected item otherwise.

Accessors. Third we implement the getters needed by the layout.

```
MethodsBrowser>>methodsList
    ^ methodsList
```

```
MethodsBrowser>>text
    ^ text
```

Layout Specification. Fourth we specify a presentation, by defining a method at the class side.

```
MethodsBrowser class>>spec
    <spec: #default>
    ^ SpecLayout composed
        add: #methodsList origin: 0@0 corner: 1@0.5;
        add: #text origin: 0@0.5 corner: 1@1;
        yourself
```

In addition of showing that the reuse of a Spec model is transparent with regard to the layout, the above method also shows that a position can be specified for each sub-model. Here, methodsList will be displayed in the top-most half of the generated widget while text will be displayed in the bottom-most half of the generated widget.

Since most of the UI elements can be expressed in terms of rows and columns, a simpler way to describe UI elements is also available. The following code produces the exact same layout as previously, in a more concise and more readable way.

```
MethodsBrowser class>>spec
    <spec: #default>
    ^ SpecLayout composed
       newColumn: [:c |
           c
             add: #methodsList;
             add: #text ];
        yourself
```

The following snippet opens the widget and populates the list with the methods of Object:

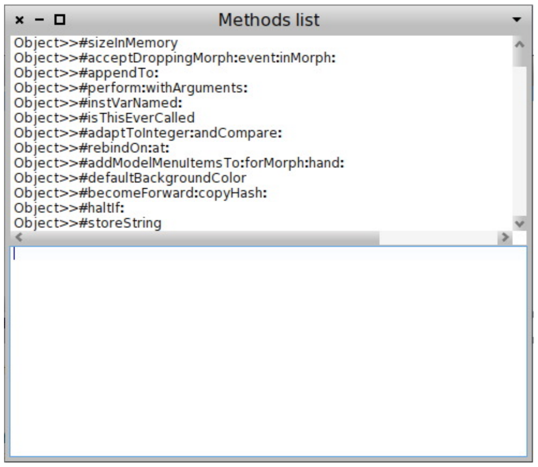
```
(MethodsBrowser new
    openWithSpec;
    yourself)
    methodsList items: Object methods
```

Figure 4 shows the result of executing the above code.

Window Title. When an item is selected the title is not updated as it used to be in MethodsList. This is addressed by implementing a title method, and slightly modifying the initializePresenter method as follows:

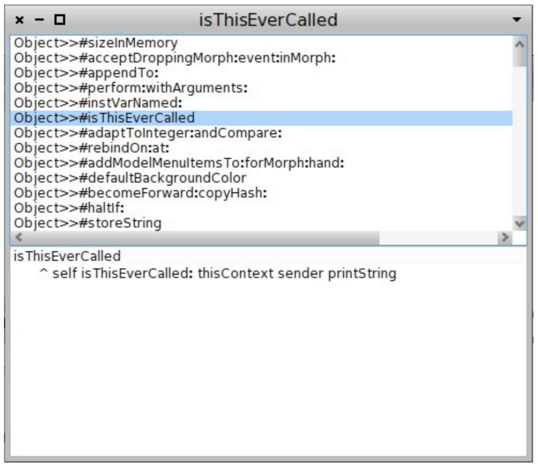
```
MethodsBrowser>>title
    ^ methodsList title


MethodsBrowser>>initializePresenter
    methodsList whenSelectedItemChanged: [:method |
        self updateTitle.
        text text: (method
                    ifNil: [ '' ]
                    ifNotNil: [ method sourceCode ]) ]
```

This gives us a dynamic title, as shown in Figure 5.



```
Object>>#sizeInMemory
Object>>#acceptDroppingMorph:event:inMorph:
Object>>#appendTo:
Object>>#perform:withArguments:
Object>>#instVarNamed:
Object>>#isThisEverCalled
Object>>#adaptToInteger:andCompare:
Object>>#rebindOn:at:
Object>>#addModelMenuItemsTo:forMorph:hand:
Object>>#defaultBackgroundColor
Object>>#becomeForward:copyHash:
Object>>#haltIf:
Object>>#storeString
```

Figure 4. The methods browser



```
Object>>#sizeInMemory
Object>>#acceptDroppingMorph:event:inMorph:
Object>>#appendTo:
Object>>#perform:withArguments:
Object>>#instVarNamed:
Object>>#isThisEverCalled
Object>>#adaptToInteger:andCompare:
Object>>#rebindOn:at:
Object>>#addModelMenuItemsTo:forMorph:hand:
Object>>#defaultBackgroundColor
Object>>#becomeForward:copyHash:
Object>>#haltIf:
Object>>#storeString
```

```
isThisEverCalled
    ^ self isThisEverCalled: thisContext sender printString
```

Figure 5. The methods browser with the dynamic title back

Public API. Recall that to have a reusable model, we need to define its public API. We will reuse this model in our last example, therefore below we show the methods of the public API for our MethodsBrowser. As these are straightforward we do not discuss them in more detail.

```
MethodsBrowser>>items: aCollection
   methodsList items: aCollection

MethodsBrowser>>displayBlock: aBlock
   methodsList displayBlock: aBlock

MethodsBrowser>>resetSelection
   methodsList resetSelection

MethodsBrowser>>selectedItem
   methodsList selectedItem
```

Note that the methods we invoke here are part of the public API we defined at the end of section 3.1.

Conclusion. This concludes the construction of our methods browser. In the construction of this browser we have shown how we can straightforwardly reuse existing models and how a selection in one widget can be used to update the display in another widget, effectively linking widgets together.

3.3 Classes Browser

The last example we provide shows how to parametrize the reuse of models, and how a UI communicates with the underlying model. We do this by illustrating how to build a simple class browser that reuses the MethodsBrowser. Being able to deeply parametrize models allows for extended reuse possibilities of the models since they can be more generic.

The five construction steps essentially are still the same, therefore we first only show a overview of the browser construction process. We will instead focus on how the reuse of user interface specifications can be parametrized. Second we modify the behavior of the reused UI model, and third modify the layout of the reused models. Fourth and last we show how to connect the browser to the class structure, i.e. to its model.

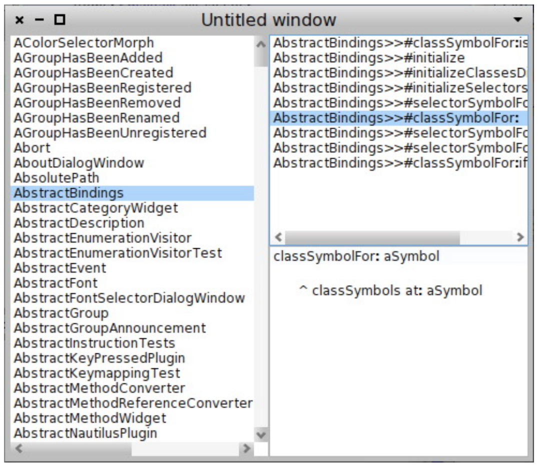
Basic Reuse. The class for the browser is called ClassesBrowser and it has two instance variables: *classes* for the list of classes, and *methodsBrowser* for the list of methods and the text zone, i.e. the methods browser we constructed above. The layout specification below shows how the two are laid out.

```
ClassesBrowser class>>defaultSpec
   <spec>
   ^ SpecLayout composed
        newRow: [:r |
          r
             add: #classes;
             add: #methodsBrowser ];
      yourself.
```

For brevity, we omit the methods that set the title, the accessors and the initialization methods that instantiate the widgets and link them together. The below snippet pops up the window shown in Figure 6 and populates it with all the classes of the system.

```
(ClassesBrowser new
      openWithSpec;
      yourself)
      classes items: Smalltalk allClasses
```

UI Parametrization. We now present how to parametrize the behavior of reused UI models. When reusing a UI model, the reusing UI model can override all parametrization parameters, *e.g.,* the displayBlock, of the reused UI model. This is done by simply providing a new parameter for the reused UI model. It overrides any parametrization made inside of the reused UI model.



```
Untitled window

AColorSelectorMorph                       AbstractBindings>>#classSymbolFor:is
AGroupHasBeenAdded                        AbstractBindings>>#initialize
AGroupHasBeenCreated                      AbstractBindings>>#initializeClassesD
AGroupHasBeenRegistered                   AbstractBindings>>#initializeSelectors
AGroupHasBeenRemoved                      AbstractBindings>>#selectorSymbolFc
AGroupHasBeenRenamed                      AbstractBindings>>#classSymbolFor:
AGroupHasBeenUnregistered                 AbstractBindings>>#selectorSymbolFc
Abort                                     AbstractBindings>>#selectorSymbolFc
AboutDialogWindow                         AbstractBindings>>#classSymbolFor:if
AbsolutePath
AbstractBindings
AbstractCategoryWidget
AbstractDescription
AbstractEnumerationVisitor
AbstractEnumerationVisitorTest            classSymbolFor: aSymbol
AbstractEvent
AbstractFont                                ^ classSymbols at: aSymbol
AbstractFontSelectorDialogWindow
AbstractGroup
AbstractGroupAnnouncement
AbstractInstructionTests
AbstractKeyPressedPlugin
AbstractKeymappingTest
AbstractMethodConverter
AbstractMethodReferenceConverter
AbstractMethodWidget
AbstractNautilusPlugin
```

Figure 6. Classes browser

In our example, when a class is selected in the classes browser, its methods are displayed. But they are displayed following the form class name»selector, where the class name is redundant as it is already given by the list selection. Also, the title shown is not correct: it displays the default title, 'Untitled window', even when items in the method list are selected. To perform these two parametrizations, we modify the initialization methods of ClassesBrowser to override the behavior specified in MethodsBrowser. The modifications consist in adding one line to the method, as shown below.

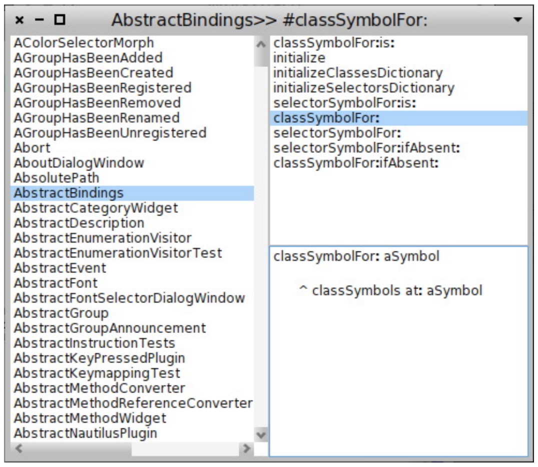
```
ClassesBrowser>>initializeWidgets

"... omitting code for behavior shown previously ..."
    methodsBrowser displayBlock: [:method | method selector ].

ClassesBrowser>>initializePresenter

"... omitting code for behavior shown previously ..."
    methodsBrowser whenSelectedItemChanged: [ self updateTi-
tle ].
```

Recall that these methods we are calling are part of the API we defined at the end of Section 3.2.

The default behavior of Spec is to ignore the window title logic of reused models, hence the presence of the default title. The last line above configures the reused method browser to update the title on a list (de)selection, using the title method defined in the classes browser. We therefore also have to implement ClassesBrowser»title. The following implementation inspects the selection of method browser to return the appropriate value. It returns 'Classes Browser' if nothing is selected, or the selected class name if only a class is selected, or class name »selector if both a class and a method are selected.

```
ClassesBrowser>>title

    ^ classes selectedItem
        ifNil: [ 'Classes Browser' ]
        ifNotNil: [:class | methodsBrowser selectedItem
                    ifNil: [ class name ]
                    ifNotNil: [:method | class name, '>> #', method se-
lector ]].
```

As shown in Figure 7, the methods are now displayed using only their selector and the title follows the form class name »selector, since a method is selected.



```
AbstractBindings>> #classSymbolFor:

AColorSelectorMorph                       classSymbolFor:is:
AGroupHasBeenAdded                        initialize
AGroupHasBeenCreated                      initializeClassesDictionary
AGroupHasBeenRegistered                   initializeSelectorsDictionary
AGroupHasBeenRemoved                      selectorSymbolFor:is:
AGroupHasBeenRenamed                      classSymbolFor:
AGroupHasBeenUnregistered                 selectorSymbolFor:
Abort                                     selectorSymbolFor:ifAbsent:
AboutDialogWindow                         classSymbolFor:ifAbsent:
AbsolutePath
AbstractBindings
AbstractCategoryWidget
AbstractDescription
AbstractEnumerationVisitor
AbstractEnumerationVisitorTest            classSymbolFor: aSymbol
AbstractEvent
AbstractFont                                ^ classSymbols at: aSymbol
AbstractFontSelectorDialogWindow
AbstractGroup
AbstractGroupAnnouncement
AbstractInstructionTests
AbstractKeyPressedPlugin
AbstractKeymappingTest
AbstractMethodConverter
AbstractMethodReferenceConverter
AbstractMethodWidget
AbstractNautilusPlugin
```

Figure 7. Classes browser with methods displayed using their selector

Layout Specification. The current layout of the ClassesBrowser view is somewhat unusual. The classic layout for such a tool is to have the two lists in a row in the top-most half and the text zone in the bottom-most half. To do this, a new layout is provided by implementing a new spec method, and set as the default spec:

```
ClassesBrowser class >>moreClassicalSpec
    <spec: #default>

    ^ SpecLayout composed
        newColumn: [:c |
          c
            newRow: [:r |
              r
                add: #classes;
                add: #(methodsBrowser methodsList) ];
            add: #(methodsBrowser text) ];
        yourself
```

In the above code, instead of using the *methodsBrowser* layout we define precisely how we want the sub-widgets to be rendered.

The result is the widget shown in Figure 8: a classes browser with the two lists in the upper part and the text zone in the lower part.

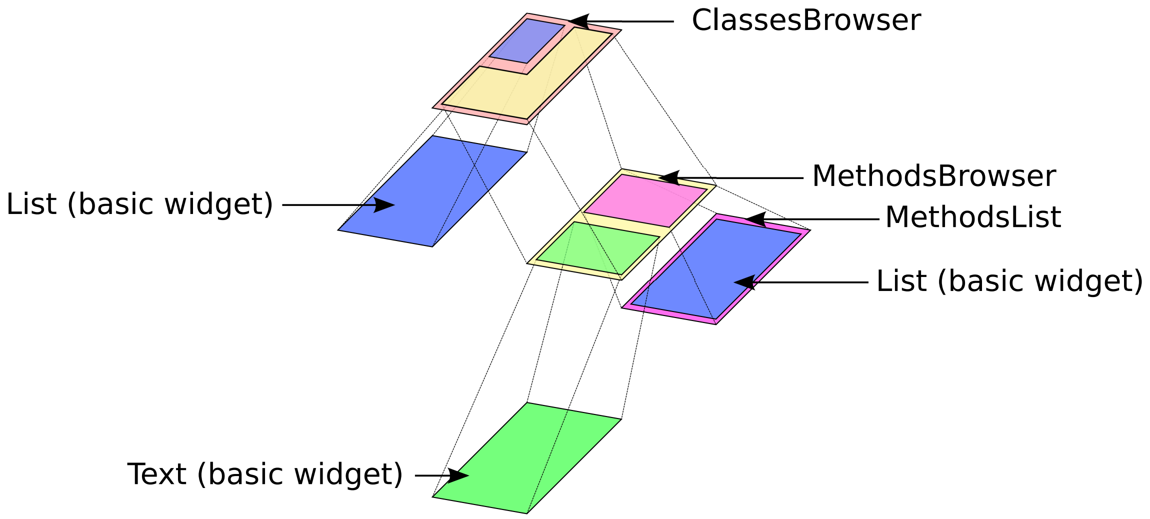


Figure 9. The three levels of layers composing the ClassesBrowser UI

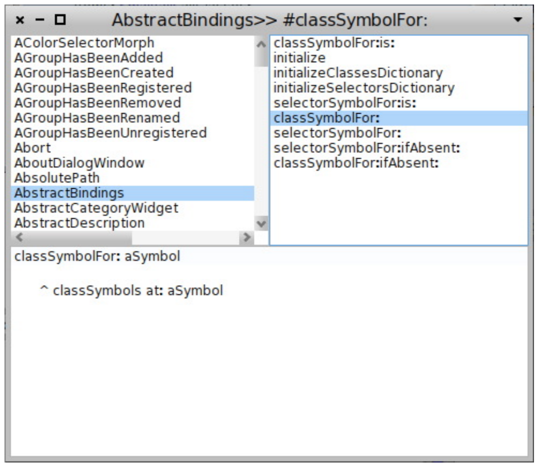

```
AbstractBindings>> #classSymbolFor:

AColorSelectorMorph            classSymbolFor:is:
AGroupHasBeenAdded             initialize
AGroupHasBeenCreated           initializeClassesDictionary
AGroupHasBeenRegistered        initializeSelectorsDictionary
AGroupHasBeenRemoved           selectorSymbolFor:is:
AGroupHasBeenRenamed           classSymbolFor:
AGroupHasBeenUnregistered      selectorSymbolFor:
Abort                          selectorSymbolFor:ifAbsent:
AboutDialogWindow              classSymbolFor:ifAbsent:
AbsolutePath
AbstractBindings
AbstractCategoryWidget
AbstractDescription

classSymbolFor: aSymbol

    ^ classSymbols at: aSymbol
```

Figure 8. Classes browser with a classic layout

Figure 9 exposes a view of the different layers comprising the ClassesBrowser UI.

Connecting the UI to the Underlying Model. As is, the UI we built is disconnected from the model it is representing: after we pass it an initial list of classes we are not able to modify it, *e.g.,* by editing methods, and it is not aware of any modifications made to its model by other browsers. We now show how to connect the UI to the underlying model, by specifying menu actions to enable the former, and subscribing to announcements to implement the latter. As this behavior belongs in the methods browser, we specify it there, and it will automatically also be present in the classes browser.

The text widget present in the methods browser provides for a standard code editing menu, where we can specify the action to take whenever the code is saved or accepted. This is performed by providing an accept block, as illustrated below:

```
MethodsBrowser>>initializePresenter
   "... omitting code for behavior shown previously ..."
   self text acceptBlock: [ :text :notifier | ... compile the text ... ].
```

The code obtains the text field from the methods browser and specified that the code, contained in the text parameter should be compiled. (We do not include the compilation code as it is not pertinent to this discussion.)

Lastly, to have the browser to react to changes in the underlying class structure, we use the system announcements mechanism to refresh its contents when needed. We rely on the fact that when such changes occur, the system raises an announcement, and subscribe to these announcements, for example using the code below the browser becomes aware of methods being added, adding them to the list of methods shown.

```
MethodsBrowser>>initializePresenter
   "... omitting code for behavior shown previously ..."
   SystemAnnouncer announcer weak
      on: MethodAdded send: #methodAdded: to: self.

MethodsBrowser>>methodAdded: anAnnouncement
   | sel text it |
   text := self text pendingText.
   sel := self methodsList selectedItem.
   it := anAnnouncement item.
   self items: (self methodsList: listItems add: it; yourself).
   self methodsList setSelectedItem: sel.
   self textModel pendingText: text.
```

The methodAdded: method first keeps a copy of the text being shown in the text field, as it may contain edits that have not been saved. It then obtains the selected item in the methods list, and adds the new method to the list. As the change in list items may change the selection in the list, it then sets the selected item to the previously stored value. Lastly, it sets the text being shown in the text field to the value stored in the first step.

Conclusion. This concludes the construction of our last example: a methods browser. In this section we have seen how it is possible to parametrize both the behavior and layout of UI models that are being reused. This allows for more generic models to be reused and customized further when needed, increasing their reuse possibility. We have also shown how to connect the user interface to an underlying model, yielding a fully functional UI.

4. The Implementation of Spec

The implementation of Spec is based on two pillars: the presence of a SpecLayout and the use of value holders. The SpecLayout is used to describe how the graphical elements are positioned inside the generated UI, by using either basic widgets or by reusing composed widgets. The value holders are used to store the variation points of the model and to react precisely to one of these changes. This way the UI updates are less frequent and more precise, and hence faster.

In this section we will discuss these two points in more detail, firstly talking about SpecLayout and secondly discussing the value holders.

4.1 SpecLayout and its Interpreter

The SpecLayout describes and represents the layout of the UI elements in Spec. More details about how to manipulate SpecLayout objets will be provided in Section 3.1. A Spec interpreter is used to build a UI widget from this layout. To have a static structure which can be easily used by the interpreter, the SpecLayout object is converted to an array before it is passed to the interpreter.

We have only seen here the composed layout, consisting of an aggregation of different Spec widgets. However there are additional kinds of layouts provided: one kind for each type of widget. This allows one to attach specific behavior to each spec type, *e.g.,* the class of the UI element specific to this kind of layout. The array representation of a spec has as first element an identifier of its type.

By using a spec type, default behavior can be defined and shared among all the widgets. It means that all the widgets produced by a defined spec type can be changed by modifying the spec type itself. Types also allow the creation of a data structure representing the tree of sub-widgets and to use a visitor pattern to build tools on top of specs, *e.g.,* like a UI builder.

The remainder of the array representation of a spec can be seen as a stack: each time a selector is read, as many arguments as needed by the selector are popped and analyzed. The spec interpreter iterates over a SpecLayout array and builds each part of the widget by recursively interpreting each element.

A SpecLayout allows one to specify where to position a widget's sub-widgets, and also allows one to position the sub-widgets of sub-widgets. This provides a way to reuse generic widgets while being able to customize them sufficiently to make them conform to a new usage scenario.

4.2 Value Holder

A value holder is a simple object which wraps a value and raises an announcement when this value is changed. Thanks to this mechanism we can use value holders as wrappers around model values and make the UI react at specific changes to the model. As such, we provide an event based structure that allows one to react to only to value change of interest.

The above means that, for example, the selection index of a list is stored in a value holder. When a new item is selected, the index changes, and the corresponding value holder raises an announcement. The basic widgets of Spec provide as part of their API event registration methods, which allow a user-defined object to react to this change if needed.

Moreover, having a value holder for each model data allows one to update the UI only for the data which has changed, without having to examine this change to establish its relevance as the DependencyTransformer in VisualWorks. This is in contrast to classical MVC [5] and MVP [10]. Here, when the observable object has changed, it sends an `update:` message to all observer objects with the changed value as argument. Then in the `update:` method, the observer has to examine the argument to react in accordance with the change. In Spec, the observer registers to each observables' value holder it is interested in, and for each value holder specifies a method to invoke when the value holder is changed. Hence the examination of the updated value is no longer necessary and the dispatch to the appropriate update logic is done naturally without any switch case.

In addition, since the whole event flow is controlled and propagated through value holders, Spec does ensure that there are no event loops due to circular event sends.

Note that since every object can register to a value holder changes, this means that a model can register itself to any of it sub-widgets value holders, or any sub-widgets sub-widgets value holder. Thanks to this, a model can add new behavior for its sub-widgets. This provides a way to reuse generic widgets while being able to parametrize them enough to make them correspond to a new scenario.

5. The spec of Spec

In this section we summarize the specification of the public APIs of the relevant building blocks for a user of Spec: the basic widgets and `SpecLayout`.

Selector	Result
displayBlock:	set the block used to produce the string for displaying items
items:	set the contents of the list
resetSelection	unselect selected items
selectedItem	return the last selected item
whenSelectedItemChanged:	set the block performed when the selected item changed

Table 1. ListComposableModel public API

Selector	Result
accept	force the text zone to accept the pending text
acceptBlock:	set the block to perform when the pending text is accepted (saved)
text:	set the text of the widget to the value provided as argument
whenTextIsAccepted:	set a block to perform when the text is accepted
whenTextChanged:	set a block to perform when the text has changed

Table 2. TextModel public API

5.1 Models public API

To build a UI the user combines basic UI models and existing Spec models as required. For Spec there is however no distinction between these two, as basic UI models are reified as Spec models. Put differently, these basic UI models are Spec models that simply wrap the widgets that are provided by the GUI framework.

Due to lack of space, we do not provide a complete specification of the public API of all models provided by Spec (11 models, in total 228 methods). The complete API for all models is provided as part of a tech report about Spec [12]. We restrict ourselves here to the public API methods of the basic models used in this paper: ListComposableModel, shown in Table 1, and TextModel, shown in Table 2.

5.2 SpecLayout

A SpecLayout is an object used to describe the layout of the UI elements of a widget.

The SpecLayout class provides a small API (only 10 methods), shown in table 3. The *add* methods and the *newRow* and *newColumn* methods cover the bulk of the use cases: adding elements to the layout. Indeed, as we have seen in Section 3 they are the only methods used when the layout is a composed layout.

The remaining two *send* methods are required to be able to interact with basic widgets. Since the Spec reification of basic UI models provides a bridge between Spec and a graphical library, the class of the UI element nor its API can be predicted. Hence we need to be able to send any message to those classes through the SpecLayout. To allow for this, the SpecLayout provides for the *send* methods, which enable performing any selector with the corresponding arguments. Thanks to these methods we ensure that a bridge can be built between Spec and any graphical library.

As an example use of the send:withArguments: method, the following code is the implementation of TextModel class»defaultSpec, which defines the binding between Spec and the Morphic UI framework for the TextModel widget. (Due to the low-level nature of this code we do not explain its functionality in detail.)

```
defaultSpec
   <spec>
   ^ SpecLayout text
         send: #on:text:accept:readSelection:menu:
         withArguments: #(model getText accept:notifying: read-
Selection codePaneMenu:shifted:);
       send: #classOrMetaClass: withArguments: #(model behavior);
         send: #enabled: withArguments: #(model enabled);
               send: #eventHandler: withArguments: #(Even-
tHandler on:send:to: keyStroke keyStroke:fromMorph: model);
         send: #vSpaceFill;
         send: #hSpaceFill;
         yourself
```

6. Spec in Pharo

Spec has been introduced in Pharo 2.0 with the goal to be used for rewriting all the tools. For now, six different widgets have been implemented:

1. MessageBrowser: a tool made for browsing messages (similar to the MethodsBrowser made in section 3.2);
2. Senders/Implementers: a tool to browse the senders or the implementors of a given selector;
3. ProtocolBrowser: a tool to browse all the methods that a given class can understand;
4. VersionBrowser: a tool used to browse the different versions of a provided method;

Selector	Result
add:	add the object provided as argument. This object can be the selector of a getter to an instance variable storing a ComposableModel or another layout.
add:origin:corner:	add the object provided as argument and specify its position as fractions.
add:origin:corner:offsetOrigin:offsetCorner:	add the object provided as argument and specify its position as fractions and offsets.
add:withSpec:	add the model provided as first argument using the selector provided as second argument for retrieving the spec. The first argument can be the selector of a method that returns a ComposableModel or a collection of such selectors.
add:withSpec:origin:corner:	add the model provided as first argument using the selector provided as second argument for retrieving the spec. and specify its position as fractions.
add:withSpec:origin:corner:offsetOrigin:offsetCorner:	add the model provided as first argument using the selector provided as second argument for retrieving the spec. and specify its position as fractions and offsets.
newColumn:	add to the current layout a column created using the block provided as argument
newRow:	add to the current layout a row created using the block provided as argument
send:	send the message with selector specified as first argument to the underlying widget
send:withArguments:	send the message with selector specified as first argument and arguments specified as second argument to the underlying widget.

Table 3. SpecLayout public API

Tool	Pharo 1.4	Pharo 2.0	Percentage of reduction
MessageBrowser	MessageSet 488	MessageBrowser 329	33%
Senders/Implementers	FlatMessageListBrowser 463	MessageBrowser + 4	99%
ProtocolBrowser	ProtocolBrowser 49	MessageBrowser + 20	47%
VersionBrowser	VersionsBrowser 312	NewVersionBrowser 57	82%
ChangeSorter	ChangeSorter (970) + DualChangeSorter (39) 1009	ChangeSorterApplication (410) + DualChangeSorterApplication (186) 596	41%
Total	2321	1006	57%

Table 4. Comparison between tools in Pharo 1.4 and Pharo 2.0

5. ChangeSorter: a tool made for managing the changes of the system;
6. DualChangeSorter: a second tool for managing changes, with focus on the transfer from one change sorter to another.

As a testament to the possibilities of reuse the MessageBrowser is used for the Senders/Implementors, and the ProtocolBrowser. Moreover the DualChangeSorter is made of two ChangeSorter linked together and specialized to add functionality involving the interactions between the two change sorters.

Table 4 shows the difference in the number of lines of code (LOC) used to implement those tools before the use of Spec (Pharo 1.4) and after (Pharo 2.0). The purpose of this table is to emphasize the reduction of code duplication. The table follows the form:

- in the first column the name of the tool which is being compared;
- in the second column the name of the class used to implement this tool in Pharo 1.4 and the number of LOC used to implement it;
- in the third column the name of the class used to implement this tool in Pharo 2.0 and the number of LOC used to implement it;
- the ratio in LOC reduction.

We will now explain the difference for each line in more details.

MessageBrowser. MessageSet is used in Pharo 1.4 to browse a collection of method references. MessageBrowser from Pharo 2.0 covers all the functionalities of MessageSet and even add new features like a topological sort or a update mechanism and the support for methods in addition of method references. Yet MessageBrowser is still smaller because thanks to widget reuse, some data of the UI itself is managed by widgets that are being reused, *e.g.,* index selection management.

Senders/Implementers. FlatMessageListBrowser is used in Pharo 1.4 to browse the senders or implementers of a selector. In Pharo 2.0 we have decided to reuse MessageBrowser since senders and implementers are also a collection of method references. MessageBrowser already covers all the FlatMessageListBrowser functionalities, and moreover adds the topological sort and the update mechanism as well. Only a trivial modification needed to be made to MessageBrowser. Hence the Senders/Implementers browser is actually a MessageBrowser, where we implemented the required API to open the corresponding list of messages. This explains why the number of line for this tool in Pharo 2.0 is so small.

ProtocolBrowser. ProtocolBrowser is used in Pharo 1.4 to browse all the methods that the provided class can understand. Again, MessageBrowser covers all the features of ProtocolBrowser and still adds the topological sort and the update mechanism. As above, MessageBrowser is reused, by adding the logic specific to the ProtocolBrowser. These 20 LOC are the algorithm to collect the relevant methods.

VersionBrowser. NewVersionBrowser provides a new tool in Pharo 2.0 that covers all the functionality of the previous tool. Implemented as its own class, it reuses MessageBrowser for the UI and beyond that only contains version retrieval methods and UI specialization methods. This leads to a low number of LOC.

ChangeSorter. The two tools have been grouped since the implementation in Pharo 1.4 moved the logic of the DualChangeSorter into the ChangeSorter class. ChangeSorter instances are aware of belonging to a DualChangeSorter or not and act accordingly.

In Pharo 2.0 the ChangeSorterApplication class is smaller than the ChangeSorter class because it only knows about itself. It doesn't contain any information about being part of a DualChangeSorterApplication or not. This is because the DualChangeSorterApplication class knows how to reuse ChangeSorterApplication and what logic to add, and as a result is bigger than the DualChangeSorter class.

But when summing up both applications, the Spec implementation is smaller even while covering all the original functionalities. This is for two reasons: firstly because checking ubiquitously for being part of a dual change sorter is expensive in term of lines of code. Secondly for the same reason than for the use of MessageBrowser, relocating UI element management to a sub-widget allows the reusing code to be concise.

Conclusion. In this section we have seen how the reuse provided by Spec is used in Pharo and how this reuse can reduced the number of lines of code (and the code duplication) by almost half. This confirms our assertion that there is a need for a declarative way to specify UIs that also allows for seamless composition and reuse of the UI declaration and logic. Moreover this shows that Spec is an effective tool to address this need. As a consequence of this observation, rewriting all the tools using Spec is a goal for the next version of Pharo.

7. Related Work

Spec is inspired by the VisualWorks [8][11] UI framework, and like it is based on static specifications *i.e.,* the SpecLayout instances at class side. In VisualWorks all the specifications are performed in terms of low level widgets which means that no composed widget can be reused. In contrast, Spec allows the reuse of high level widgets and as a result, the specifications are simpler. Thanks to this fact the UIs can be composed of smaller widgets that make the system more modular and easier to maintain.

Spec also follows the lead of Glamour [1] in favoring an event-based flow through the widgets. However, Spec can be used for every kinds of UI while Glamour is restricted to browsers. Spec widgets also explicitly declare a public API instead of heavily relying on blocks, as Glamour does.

For the UI generation part, Spec is different from tools like NetBeans [7] or WindowBuilder [3] in the sense that they both only provide graphical tools for generating user interfaces while Spec is based on a text based description of the UI. Furthermore, where NetBeans and WindowBuilder generate java code, Spec uses an object and relies on this object for describing the user interface. Instead NetBeans or WindowBuilder use an XML file or parse Java source code.

The disadvantage of this is that if the XML file is edited by hand or if some parts of the generated Java code is refactored these tools are not always able to handle these changes.

In addition of the UI code Spec also provides a framework for the model behavior when NetBeans or WindowBuilder only provide UI elements generation source code. Indeed, Spec can be used to define (and reuse) the logic links between widgets where NetBeans or WindowBuilder can only be used to generate UI elements.

XUL [6] is an XML based language used for describing and reuse widgets through *overlays*. While a group of widgets can be reused, unlike Spec XUL doesn't allow for locally changing the inner logical links. SWUL [2] is a DSL based on the strategy transformation framework that proposes a more declarative syntax for expressing widgets description in Swing. SWUL behaves like XUL in the sense of not being able to locally redefine the behavior of a sub-widget.

8. Conclusion

In this paper we presented Spec, a UI builder whose goal is to support UI interface building and seamless reuse of widgets. Spec is based on two core ideas: first the declarative specification of the visual behavior of widgets and their logic, and second inherent composability of widgets, based on explicit variation points.

In our experience maintaining Pharo, we have seen that there is a nontrivial amount of code duplication in UI code which can be avoided and that the logic of one widget is often based on the wiring of the logic of adjacent or nested widgets. Hence being able to compose and reuse existing behavior is central to be able to build new widgets.

We have shown how Spec can be used, by providing three example UIs that highlight the reuse and parametrization features of Spec. This was followed by a more formal specification of the APIs used in the example and an overview of the most relevant points of the implementation. We then showed how Spec enabled a 57% of code reduction in the re-implementation of six UIs of Pharo, thanks to a high amount of reuse of widgets.

The latter shows that Spec provides ample support for reuse of widgets and is an appropriate tool to address the problem of code duplication in UI code. As a consequence it will be the standard UI builder for Pharo 2.0 and all UI tools in Pharo will be rewritten using Spec.

Availability

Spec is part as standard of Pharo 2.0 and is also available in Pharo 1.4, its Metacello configuration is called ConfigurationOfSpec and is available from SqueakSource3 (http://ss3.gemstone.com/).

Acknowledgments

This work was performed in the context of the INRIA Associated Team PLOMO (2012).

References

[1] P. Bunge. Scripting Browsers with Glamour. Master's thesis, Fakultät der Universität Bern, April 2009. Available at: http://scg.unibe.ch/archive/masters/Bung09a.pdf.

[2] R. de Groot. Implementation of the Java-Swul language: a domain-specific language for the SWING API embedded in Java. Master's thesis, Faculty of Science, Utrecht University, January 2005. Available at: http://strategoxt.org/pub/Stratego/Java-Swul/swul-article.pdf.

[3] Eclipse Technology. Windowbuilder user guide. Technical report, Google, 2011. Available at: http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.wb.doc.user%2Fhtml%2Findex.html.

[4] T. Howard. *The Smalltalk Developer's Guide to VisualWorks*. SIGS Books, 1995. ISBN 1-884842-11-9.

[5] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, Aug. 1988.

[6] Mozilla Developer Network. XUL - MDN, 2012. Available at https://developer.mozilla.org/en/XUL.

[7] NetBeans. Netbeans IDE. http://www.netbeans.org, archived at http://www.webcitation.org/5p1qB6hNt, 2010. URL http://www.netbeans.org.

[8] ParcPlace-Digitalk. *VisualWorks cookbook*, October 1995. Available at: http://www.esug.org/data/Old/vw-tutorials/vw25/cb25.pdf.

[9] ParcPlace89. Parcplace systems, Objectworks Reference Guide, Smalltalk-80, version 2.5, chapter 36, 1989. ParcPlace Systems.

[10] M. Potel. MVP: Model-View-Presenter. the Taligent programming model for C++ and Java. Technical report, Taligent, Inc., 1996. Available at: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

[11] I. Tomek. *The Joy of Smalltalk*, September 2000. Available at: http://stephane.ducasse.free.fr/FreeBooks/Joy/6.pdf.

[12] B. Van Ryseghem. Spec – Technical Report. Technical report, Inria – Lille Nord Europe - RMoD, 2012. Available at: http://hal.inria.fr/docs/00/70/80/67/PDF/SpecTechReport.pdf.