Evaluating Java Runtime Reflection for Implementing Cross-Language Method Invocations

Stefan Sobernig
Institute for Information Systems and New Media, Vienna University of Economics and Business, Austria
stefan.sobernig@wu.ac.at

Uwe Zdun
Information Systems Institute, Vienna University of Technology, Austria
zdun@infosys.tuwien.ac.at

Abstract

Cross-language method invocations are commonly used for integrating objects residing in different programming language environments. In this experience report, we evaluate the performance and the design impact of alternative implementations of cross-language method invocations for the object-oriented scripting language Frag, implemented and embedded in Java. In particular, we compare reflective integration and generative integration techniques. For that, we present a performance evaluation based on a large set of test cases. In addition, we propose a new method for quantifying and comparing the implementation efforts needed for cross-language method invocations based on cross-language refactorings. We report on the lessons learnt and discuss the consequences of the implementation variants under review.

Categories and Subject Descriptors D.1.5 [*Software*]: Programming Techniques—Object-oriented Programming; D.3.2 [*Language Classifications*]: Object-oriented languages; D.2.8 [*Software Engineering*]: Metrics—Complexity measures, Performance measures

General Terms Languages, Measurement, Performance

Keywords Reflection, cross-language method invocation, domain-specific languages, refactoring, design science

1. Introduction

Object-oriented (OO) language systems and derivatives thereof, such as component-oriented systems and systems built using distributed object systems, regularly need to be integrated. In all of these cases, forms of language interoperation are needed. Typical scenarios of language interoperation are, for instance, the two-way interactions between language-specific components written in two different languages (e.g., using Java and C++ in one system [34]), multi-language virtual machines [11, 28], and implementations of one language by means of another. This is the case for dynamic, scripting, and domain-specific languages (DSLs) which are implemented on top of host languages such as Java, C#, C++, or C.

In this paper, we are concerned with the latter area of language interoperation: We provide an empirical evaluation of integrating two object-oriented languages with one serving as the host (implementing) and with the other being the embedded (implemented) language. When creating an embedded language, its core and its runtime environment are implemented in the host language. As our evaluation artifact, we review the case of the Frag language [13]. Frag is a dynamic, object-oriented scripting language implemented in Java, and it provides for developing internal and external DSLs on top of it [35]. While, in general, this kind of language integration is motivated by providing seamless access to components implemented in the host language, the Frag case presented here is also about facilitating the process of refactoring embedded into host code, especially in support of DSL prototyping.

Integrating two OO languages can cover different kinds of language features to varying extents. In particular, two language models and execution environments must be aligned [11, 15, 28]. This includes, for instance, different schemes of object creation and object life-cycling, object-type handling and type systems, divergent kinds of classification and inheritance relationships, as well as differences in message passing and distinct reflection facilities. In the following, we limit ourselves to the issue of *cross-language method invocations* [11, 29]. That is, we put emphasis on evaluating design choices when bridging message passing, method lookups, and method dispatching between the embedded and the host language. For implementing cross-language method invocations, forms of *reflective* and *non-reflective* integration are available. For the scope of this paper, reflective integration means runtime reflection. More precisely, we look at the Java-specific forms of runtime introspection and meta-level reification, as available through the Java Reflection API. By non-reflective integration we refer to *generative* integration techniques. These include kinds of static code generation, i.e., source-to-source transformation, while excluding runtime kinds of code generation [7]. We selected Frag [13] because it uses cross-language method invocations in a ubiquitous manner, turning the adoption of appropriate Java implementation techniques into a critical design decision.

Whatever features are aligned and integrated, language interoperation aims at representing data structures of one language in the realm of the other and at calling from one language into the other. Ideally, cross-language data representation and cross-language instruction calling are implemented in a symbiotic manner [15]. This requires the data and protocol integration between two languages to be transparent (e.g., avoiding explicit, language-level wrapper structures) and symmetric (e.g., transparent method calls are available in both directions). Also, if available at all, a language's meta level is to be exposed to the other language (e.g., message reification as provided in some meta-object protocols). The integration scenario studied in this paper is partially symbiotic. We focus on one-way, asymmetric cross-language method invocations, that is, invocations from within the embedded into the host language and

PPPJ '10, September 15–17, 2010, Vienna, Austria.
Copyright © 2010 ACM 978-1-4503-0269-2...$10.00

not vice versa. Also, we do not reflect on meta-level integration for cross-language method invocations. That is, we do not expose reifications as provided by the Java Reflection API (e.g., instances of the `java.lang.reflect.Method` class) to the embedded language.

Despite our not perfectly symbiotic and thus relaxed integration scenario, we could not identify substantial guidance on making design and implementation decisions for cross-language method invocations in related work on cross-language integration. From the viewpoint of developers of embedded languages or DSL developers considering a DSL development platform for adoption, there is a lack of documented and systematic evaluations of existing language designs and language implementations. This is problematic because of the considerable variety of reflective and non-reflective techniques available for implementing cross-language method invocations in Java (and beyond).

The contributions of this paper are twofold: On the one hand, we propose a new method for quantifying and comparing the efforts needed for cross-language method invocations based on *cross-language refactorings*. On the other hand, we present a performance evaluation of two cross-language method invocation implementations for the Java implementation of the Frag [13] language core. Our familiarity with Frag's implementation and the availability of a large test suite covering the entire feature set of this language (e.g., garbage collection, string and list manipulation, call-stack management) permit us to compare cross-language method invocations quantitatively in a benchmark setting. We report on the lessons learnt and discuss this approach for evaluating the adoption of different implementation variants of cross-language method invocations. Our findings shall help both developers and adopters of embedded languages to conduct similar evaluation studies in the context of their decision-making processes.

The remainder of this paper is organized as follows. Section 2 reiterates over the implementation strategies available and highlights the shortcomings of the related work when it comes to support a decision making on realizing cross-language method invocations. In Section 3, we introduce our design artifact, the Frag language. Details of the alternative frameworks for cross-language method invocation realized for two different versions of the Frag core are given. Sections 4 and 5, then, discuss the comparative evaluation based on cross-language refactorings and on profiling the runtime performance, respectively. In Section 6, we compare with related work. Section 7 summarizes major findings and we conclude in Section 8.

2. Cross-Language Method Invocations

There are different implementation options for realizing cross-language method invocations. Either, one introduces method dispatch code written or generated in the host language or one reverts to reflection-based message bridging between the embedded and the host language (see [7, 29, 34]). We briefly review these two approaches and selected implementation variants thereof. Then, we look at the state of closely related work on evaluating these options.

2.1 Implementation Options

A first approach to implement cross-language method invocations is to provide hand-written wrappers for cross-language method dispatch using a dedicated API of the host language. Such APIs offer to register the host language methods with an embedded method dispatcher and provide for basic wrapping and unwrapping of parameters and return values. Available implementation variants are dispatchers based on switch control structures (i.e., switch threading) or types of method objects. The downside of hand-written wrappers is that they require similar development actions to be performed for each and every host method that should be exposed. In addition, depending on the implementation variant, the wrapper code has to be maintained at multiple locations to manage single host methods (see Section 3 for an actual implementation).

Code generation can be used to automate these recurring tasks by transforming interface descriptions into dispatch wrappers at design (or compile) time. In the middleware context, you may think of code generators for client proxies and invokers based on interface description languages such as the Interface Description Language (IDL) or the Web Service Description Language (WSDL). In the programming language integration context this involves wrapper generators and their interface description format (such as SWIG [32]). Code generation does not need to be performed at design time only; for example, embedded compilers such as Janino [20] allow us to generate code at runtime, too. Using generated dispatch wrappers, however, requires developers to put in additional development efforts and to understand yet another language artifact ([33]). Alternatively, we might need to devise forms of static reflection (i.e., design or compile time reification and introspection of host language code) to feed code generators. For both, hand-written and generated dispatch wrappers, the runtime behaviour (e.g., regarding performance and scalability) is comparable, once the generation has been performed.

A general alternative to the above solutions are reflective wrappers. A runtime method reflection infrastructure provides application meta-data to construct and perform invocations (i.e., invocation target, parameters, parameter types, etc.) at runtime. The methods of a class are reified into runtime (meta-)objects, which themselves can receive messages. For most kinds of reflective method calls, introspective access to host methods is sufficient (as in the Java Reflection API; see Section 3). Method-reflective integration offers certain advantages. First, runtime reconfiguration becomes possible (e.g., dynamic switching between different embedded method implementations and method delegation; [7]). Second, method implementations may be injected at different binding times specific to the embedded language (e.g., module import and initialization, interpreter setup, or evaluation time of scripts). Third, reflective dispatch wrappers permit more direct access to host language constructs, avoiding glue code to wrap host methods [29]. However, reflective method calls cause a certain runtime overhead compared to non-reflective dispatch wrappers, both in terms of extra execution time and memory consumption. Depending on the implementation of reflective method invocations in the host language (e.g., "out-of-line" branching to the physical method memory locations vs. bytecode injection for JVMs [27]) and the interactions between reflective methods calls and the method dispatch scheme (e.g., static vs. dynamic dispatch), there is a penalty to the method execution time [6, 28]. These timing costs relate to reification (i.e., method and meta-data look-up, generating a reflection wrapper object), invocation setup (i.e., bytecode or machine code generation and injection), parameter handling (in particular, signature adaptation involving representation mapping and type conversions), and invocation processing (i.e., method access restrictions are checked based on the caller object). In addition, such a reified layer of indirection undermines certain forms of runtime optimization. As for the JVM, performing inline caching is hindered by, e.g., indirection calls clouding inlining prediction and by static references to the wrapper objects ruling out inline caches (see [28] for details).

2.2 Motivating Problems in Detail

Unfortunately, in a design project such as the Frag language, it turns out difficult to decide for a cross-language method invocation variant from an analytical evaluation based on literature reviews and on harvesting related language implementations. This is because the consequences of such a design decision for a language system as a whole remain unclear. The closely related work presents the following shortcomings:

- *Missing or incomplete documentation of design decisions and quantitative evaluations*: The designs of embedded languages with respect to cross-language method invocations are largely undocumented, especially the underlying rationale of the decision-making. While reflective mechanisms are widely used, these uses also include aspects such as cross-language object generation, as well as cross-language interface and implementation inheritance [5, 11]. Little has been reported on cross-language message passing and method invocations as such. Quantitative design evaluations (e.g., quantification of runtime profiles) are often incomplete and are not reproducible (see, e.g., [5]). This limits the applicability of such findings to informed design decisions in other development projects.
- *Lopsided design evaluations*: The usage of reflection in cross-language designs is only considered in terms of its time and space overhead (see, e.g., [5, 9]). An integrated view of requirements and the forces between these requirements and performance costs are not discussed. Possible requirements are the lack of runtime reflection infrastructures or code refactorings after phases of rapidly prototyping domain-specific languages (DSLs).
- *Only micro-profiling of performance overhead*: Embedded languages based on reflection and reflective wrappers are commonly evaluated based on atomic and synthetic execution time profiling and benchmarking. This includes control structure performance [23] and limited method dispatch scenarios (e.g., throughput measurements for empty methods; [5]). Results from macro-profiling studies, which are based, for instance, on test suites covering entire features of the embedded language (e.g., garbage collection), are not available. The relative reflection overhead describes the time costs created by a reflective method setup and dispatch relative to the time costs of the core behaviour implemented by a method. Macro-profiling, however, would allow to assess the relative overhead of reflective techniques in the context of the embedded language execution environment.
- *Impact of the embedded language model on assessing reflection*: The reflection overhead must be considered in view of the method binding and object lifetime model as implemented by the embedded language. Reflective method invocations often align to the lifecycle of the embedded objects and their methods so that major overheads do not become effective at critical times, i.e., the actual method invocation. Hence, the overhead implied for the whole system remains unclear. This aspect is not covered by the related work under review.

3. The Case of the Frag Language

In this paper, we report on a design science research project [17] in which we have built a language infrastructure for dynamic programming and domain-specific language (DSL) development, called Frag [13]. Frag is implemented in Java. Cross-language method invocations are an important aspect of Frag because they facilitate rapid prototyping of DSLs. In many cases, a DSL is first implemented as a Frag script and, as the implementation matures, it is incrementally moved to a Frag/Java implementation (e.g., to improve its performance profile). This kind of rapid prototyping is hindered by the risk of a high-effort refactoring into Frag/Java. This risk is caused by a possible lack of tool support for automated Frag/Java code generation and, therefore, by the need for excessive hand-coding. Hence, the flexibility and the efforts required for refactoring Frag into their Frag/Java equivalents was a major design issue. In our decision-making process for the Frag language design we faced two design questions, which were difficult to answer from a literature and implementation review alone (see also Section 2):

1. How can we minimize the developer effort needed to implement a cross-language method invocation when a scripted Frag object is refactored into a compiled Frag/Java object?
2. Provided that a reflection-based approach is adopted, what is the trade-off in terms of the runtime performance due to the reflective method calls?

For the remainder of this section, consider a simple Frag script. This code snippet illustrates declaring a class `AnObject` and an operation `foo` owned by this class:

```
# 1. Defining a Frag class 'AnObject'
Object create AnObject
# 2. Defining a method on the class 'AnObject'
AnObject method foo args {
  # ... do something ...
}
# 3. Creating an instance of the class 'AnObject'
AnObject create anInstance
# 4. Sending a message 'foo' to the instance
anInstance foo
```

This scripted Frag object is now to be refactored into an equivalent Frag object definition in Java. That is, while the method use (i.e., the invocation of `foo`) remains unchanged, the method definition of `foo` is to be realized in Java. Porting the method definition to Java should be transparent to the Frag developer, i.e., referring to and using `AnObject` should be possible as it was defined at the script level alone, without causing changes to its observed behaviour. Looking at the above listing, creating `anInstance` and sending a message `foo` to this object should still be realizable as shown. However, behind the scenes, the object handle `AnObject` and the message `foo` must be matched by corresponding Java structures, i.e., the message receiver would be an instance of a Java class providing a method definition for `foo`. This is a case in point for a cross-language Frag-to-Java method invocation.

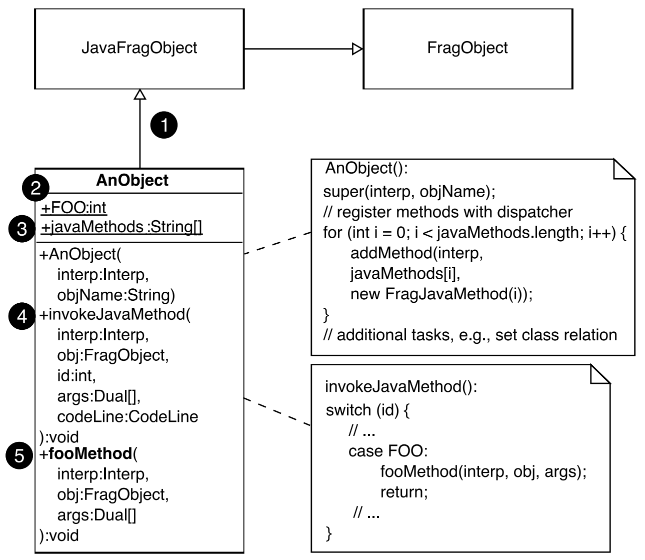
To address the two design questions raised above, we implemented two different Frag-Java integration schemes in the Frag runtime environment. For Frag 0.6, we employed switch-threaded, non-reflective Frag/Java method wrappers (see Section 3.1). In Frag 0.7, we adopted Frag/Java method wrappers based on Java method introspection (see Section 3.2). In the following, we introduce these two implementation variants in more detail.

3.1 Switch-Threaded Wrapper

The first alternative is the integration based on a switch-threaded, non-reflective wrapper. Figure 1 shows this implementation variant when used for refactoring `AnObject` into its Frag/Java equivalent. To realize a wrapper with one method, a Java wrapper class and a set of auxiliary structures must be created. This structure has been chosen because it can easily be generated from an interface description. The wrapper class refines the `JavaFragObject` class (1), the general superclass of all classes wrapping Java code exposed in Frag. An `int` constant (`FOO`) is used to realize the switch block (2). In the `javaMethods` array such constants are mapped to Java method names (3). The Frag dispatcher resolves a method identifier (i.e., the method name `foo`) to the switch constant via this method array. The resolved index is given to the dispatch handler, i.e., `invokeJavaMethods` (4), which invokes the corresponding Java method (e.g., `fooMethod`) (5). The Java method obtains invocation and context information from the Frag dispatcher, i.e., the parameter list, the interpreter object, and the receiving Frag object.

3.2 Method-Introspective Wrapper

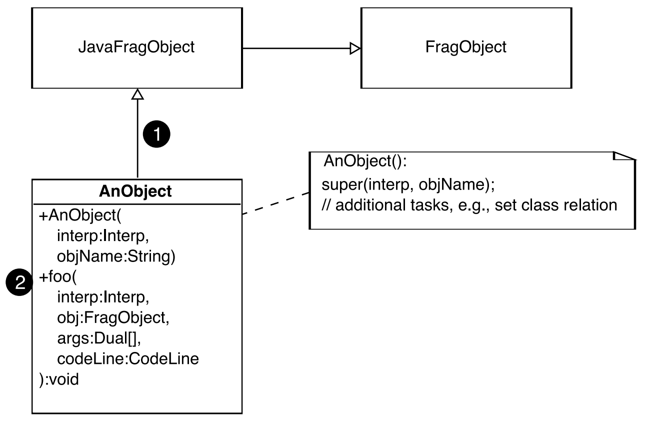
The second implementation variant is the reflective integration based on method-reflective wrappers. This variant is based on the



```
AnObject():
super(interp, objName);
// register methods with dispatcher
for (int i = 0; i < javaMethods.length; i++) {
    addMethod(interp,
    javaMethods[i],
    new FragJavaMethod(i));
}
// additional tasks, e.g., set class relation
```

```
invokeJavaMethod():
switch (id) {
    // ...
    case FOO:
        fooMethod(interp, obj, args);
        return;
    // ...
}
```

Figure 1. Switch-threaded, non-reflective implementation of a Frag/Java object



```
AnObject():
super(interp, objName);
// additional tasks, e.g., set class relation
```

Figure 2. Method-reflective implementation of a Frag/Java object

Java Reflection API, i.e., the `java.lang.reflect.Method` class. This Java built-in form of runtime reflection is based on the reification of Java language elements (e.g., methods) as runtime objects, which can be interacted with through method invocations [6]. Internally, intercession is achieved by generating and injecting bytecode [27].

Figure 2 depicts the Frag/Java definition of the previously scripted `AnObject` in Frag 0.7. The wrapper class (`AnObject`) is a subclass of `JavaFragObject` (1), which provides the same services as in the non-reflective implementation. However, it was assigned the responsibility of registering and reifying Java methods to be exported as Frag methods at the setup time of the wrapper class. The possible setup times correspond to the initializations phase of the Frag interpreter and to later, on-demand class loading. The Java method (e.g., `foo`) to be accessible as a Frag method is automatically and directly exposed using the Java method name by convention as the Frag method identifier (2). Hence, the Java method (due to its reification as a `java.lang.reflect.Method` instance) does not require the mapping between an external handle and an internal method call. Also, the wrapper-specific dispatch handler (i.e., `invokeJavaMethods`) turns obsolete.

4. Refactoring Complexity

To compare the two implementation variants for Frag-to-Java method invocations regarding their design and refactoring complexity, we introduce the notion of *cross-language refactoring*. We report on four examples of cross-language refactorings which were frequently observed in the Frag rapid prototyping context. Then, we quantify the transformation steps involved in these four cross-language refactoring scenarios for both the non-reflective and the reflective implementation approaches.

4.1 Cross-Language Refactorings

A refactoring is a principled modification to a program's code representation without changing its external behaviour with the objective to improve the program's design [12]. A refactoring is characterized by a number of behavioural invariants (to be preserved by a refactoring) and a set of source-code transformations. A cross-language refactoring [24] translates single or collaborations of objects of the embedded, interpreted language (i.e., Frag) into their behaviourally equivalent implementations in the host language (or, vice versa). In this evaluation study, we are interested in converting implementations in the Frag embedded language into Java. This kind of cross-language refactoring follows various objectives:

- Frag programs as results of rapid prototyping (e.g., in the DSL context) turn mature and their production-grade deployment requires the performance increment offered by a host language implementation of their behaviour.
- Scripted language extensions are considered for integration into the core of the embedded language. This adoption into the language core is facilitated by providing the extension implementation in the host environment. Lower-level facilities are made available to the extension developer as well as points of introspection and of intercession can be specified.
- Functional extensions to the library (e.g., application frameworks for I/O and network connectivity) have been scripted based on platform-level controls offered by the language core. Now, specialized and highly optimized components in the host language are considered for integration. The scripted Frag extension is then turned into a mere wrapper around these components, and we aim at providing a Frag language binding to this Java component.

We consider different kinds of refactorings in the context of our Frag experience. Partly, they have been reported as relevant refactoring in object-oriented designs in general [12]. We observed the following refactorings for cross-language method invocations to be frequently used in Frag rapid prototyping projects:

- REPLACE EMBEDDED WITH HOST OBJECT: Adding a Frag/-Java class definition to re-implement the behaviour described by a scripted Frag object.
- REPLACE EMBEDDED WITH HOST METHOD: Realizing Frag methods as Frag/Java methods owned by a Frag/Java class.
- RENAME HOST METHOD: Renaming a Frag/Java method on a Frag/Java class.
- REMOVE HOST METHOD: Removing a Frag/Java method from the owning Frag/Java class.

4.2 Refactoring Complexity in Frag

Table 1 provides an overview of the transformation complexity caused by each of the two integration strategies. Figure 3 shows an example specification of the 8 transformations needed for REPLACE EMBEDDED WITH HOST METHOD for the non-reflective case, as well as the 2 transformations needed for the reflective case. We have compared all other refactorings in the same manner.

Refactoring	Switch-threaded wrapper	Method-introspective wrapper
REPLACE EMBEDDED WITH HOST OBJECT	7	3
REPLACE EMBEDDED WITH HOST METHOD	8	2
REMOVE HOST METHOD	5	1
RENAME HOST METHOD	4	1

Table 1. Number of transformation steps needed under either integration strategy

Non-reflective integration (8 transformation steps):

1. Create (or duplicate) a constant field on the target `JavaFragObject`.
2. Rename the constant to reflect the intended method name (e.g., `FOO` for the method name `foo`).
3. Increase the enumeration index number assigned to the new constant field. Make sure it matches the method name's position in the `javaMethods` array.
4. Insert the method name string (e.g., `foo`) into the `javaMethods` string array.
5. Add a method-specific case statement to the switch block in `invokeJava-Methods`. Use the constant field as matching expression for the case branch.
6. Create (or duplicate) the skeleton for the implementing Java method.
7. Rename the implementing Java method to reflect the target Frag method name (e.g., `fooMethod`).
8. Create (or duplicate and rename) the method call to the new Java method in the case branch previously inserted.

Reflective integration (2 transformation steps):

1. Create a bare Frag/Java method definition, e.g., by copying an existing method block from the owning Frag/Java object.
2. Adjust the method name.

Figure 3. A REPLACE EMBEDDED WITH HOST METHOD refactoring in Frag

For the refactoring REPLACE EMBEDDED WITH HOST OBJECT, 7 transformations are needed using the non-reflective integration case. In contrast, the reflective integration requires only 3 transformations. For the refactoring REPLACE EMBEDDED WITH HOST METHOD, we counted 8 transformations for the non-reflective compared to 2 for the reflective integration; for RENAME HOST METHOD, we counted 5 transformations for the non-reflective compared to 1 for the reflective integration; and for REMOVE HOST METHOD, we counted 4 transformations for the non-reflective compared to 1 for the reflective integration (see also Table 1).

These transformation counts illustrate that even fairly basic modifications entail relatively complex sequences of primitive refactoring steps in the cross-language setting. In particular, the non-reflective approach entails more steps than the reflective one, yielding an increased number of potential sources of failure. However, the non-reflective approach can certainly benefit from generative techniques (e.g., forms of guided code generation in the IDE). Hence, when using generative techniques, the number of transformations is only relevant for tasks such as writing transformation templates for the generator or inspecting the generated results.

Composite refactorings, which involve multiple atomic transformations, add further complexity in the non-reflective approach. An example would be the MOVE METHOD [12] refactoring, quite common in realigning object-oriented designs. Moving methods from one to another owning Frag/J̃ava object might be justified because the method in question exhibits more usage dependencies on object members of the target than the source object. Or, Frag/J̃ava methods are moved up the inheritance hierarchy in an act of generalization (i.e., a PULL UP METHOD refactoring [12]). A MOVE METHOD refactoring consists of a combined REMOVE HOST METHOD and REPLACE EMBEDDED WITH HOST METHOD refactorings, requiring at least 13 steps in the non-reflective and only 3 in the reflective integration strategy.

Let us consider a recent cross-language refactoring completed for a Frag-based component. Frag features a template editor which provides for creating authoring environments for DSLs developed on top of Frag. A component of the template editor realized in Frag was ported over into a Frag/Java class through a REPLACE EMBEDDED WITH HOST OBJECT and multiple REPLACE EMBEDDED WITH HOST METHOD refactorings. The target class owned 11 methods. According to Table 1, these refactorings required a total of 95 transformations (i.e., $11 * 8 + 7$) for the non-reflective integration compared to just 25 (i.e., $11 * 2 + 3$) in the reflective case. In other words, the transformation using the non-reflective approach takes approximately 74 % more development effort than using a reflective approach.

5. Runtime Performance

Usually, the biggest disadvantage reported for the reflective integration approach is its inferior performance (see Section 2; [6, 7]). In this section, we compare the performance of the two cross-language integration approaches. We contrasted the Frag versions 0.6 and 0.7 [13], with 0.7 adopting the reflective integration approach. For 0.6, the Java code base representing the language core amounts to 10,413 source lines of Java code (SLOC), for 0.7 to 9,579 SLOC. We adapted the two language versions for this study so that they differ only in their cross-language method invocation implementation. All other changes between Frag 0.6 and 0.7 have been factored out. The reduction in the code base size is exclusively due to the reflection-based refactoring.

The Frag project [13] is well-suited for our performance evaluations because it provides a large test suite, covering all language features. That is, in contrast to benchmarks specifically developed to measure a feature or simplistic examples, a holistic test suite offers insights on how the reflective implementation of cross-language method invocations affects the performance of the entire Frag core. Our test suite consists of 17 test components, testing different features of the language, ranging from 8 to 418 test cases each. In total, Frag comes with 1243 test cases. These are realized in 9,463 source lines of Frag code (SLOC). Each test component involves numerous method calls. While complex test cases define method call sequences, single method calls usually involve nested evaluations and string substitutions (e.g., for assembling parameter values) which, in Frag, also result in method calls.

We measured the performance on a rather weak desktop machine, as our approach will usually need to run on the local machine of developers. The machine had an Intel Core2 Duo CPU, 1.60 GHz processor with 1.96 GB RAM and was running under Windows XP. We used the Java JRE 1.6.0_07. All test runs were performed under the JVM's default memory and garbage collection configuration. Each of the 17 test components was measured 10 times. Each run was performed as a first iteration. Therefore, the execution times recorded imply JVM setup and initial compilation times. For the same reason, the timing probes do not reflect the effects of possible dynamic optimizations achievable in follow-up iterations. We report the averages and the standard deviations for these test component runs (see Table 2).

With exception of some outliers, we learn from Table 2 that the standard deviation across the test components is rather low. For the non-reflective and reflective integrations cases, the average standard deviations are 3.12 % and 2.45 %, respectively. In both cases, we observe outliers with a deviation of up to 15.12 % and 13.43 %. Overall, the reflective integration performs a little better in terms of

Test component			Switch-threaded wrappers		Method-introspective wrappers		Δ
	# test cases	# method calls	(A) Avg. test execution time (in μs)	Relative standard deviation (in %)	(B) Avg. test execution time (in μs)	Relative standard deviation (in %)	$(B - A) * 100/A$ (i.e., in %)
Basic Frag commands	42	822	66,472.70	5.61	91,992.20	1.45	38.39
Math commands	57	1,065	81,873.90	1.81	90,732.10	13.43	10.82
Expressions	418	6,462	211,581.60	2.74	251,431.80	4.48	18.83
Object system	113	4,197	139,907.10	4.09	155,564.50	1.82	11.19
Dispatchers	10	6,860	15,234.60	7.05	16,513.10	2.77	8.39
Callstack	22	4,612	15,721.70	12.26	17,253.50	4.03	9.74
Command classes	13	7,205	14,402.70	6.90	15,947.90	3.94	10.73
Exception handling	30	5,586	52,320.70	7.06	56,067.40	4.00	7.16
Hashtable	20	7,835	32,324.90	4.33	36,340.60	3.62	12.42
Garbage collection	47	10,106	128,055.40	2.97	162,561.30	2.24	26.95
Unparsed regions	8	7,956	3,338.00	6.09	3,660.90	3.10	9.67
File handling	69	11,567	311,195.30	15.12	302,227.20	9.37	-2.88
List handling	178	7,827	91,785.20	1.17	128,334.20	1.58	39.82
String handling	101	19,122	17,7061.40	0.91	17,2989.90	1.46	-2.30
Control-flow commands	82	12,511	45,127.80	2.12	50,237.80	2.67	11.32
Interpreter methods	13	19,421	12,138.10	2.05	13,192.60	2.10	8.69
Java wrapper	20	13,133	33,194.30	2.12	36,429.20	0.84	9.75
Total	**1,243**	**146,287**	**1,431,735.40**	**3.12**	**1,601,476.20**	**2.45**	**11.86**

Table 2. Performance comparison

the standard deviation, but the results are comparable. Hence, it is justified to interpret the average execution times reported.

The difference between the average numbers ranges from -2.88 % up to 39.82 %. The average delta is 11.86 %. That is, in average, a performance penalty of approximately 12 % can be expected for adopting reflection-based cross-language method invocations. However, in cases heavily dependent on the reflective integration, this penalty increases to 40 %. In seldom cases, reflective integration even shows a very small positive impact on performance. We can observe that this applies to features, such as string and file handling, which are relatively expensive compared to basic language features. In contrast, when mainly base language features are used, the performance costs of reflective calls are significant. We can draw two conclusions from these data:

1. Given the positive effect on cross-language refactorings, the reflective approach should be used for rapid prototyping projects if modest performance degradation is tolerable.
2. For performance-critical parts of an application, falling back to non-reflective cross-language method invocations remains a viable option.

Besides, we used these profiling results to improve Frag's performance profile. In the upcoming Frag release, the test components suffering from considerable performance degradation, namely garbage collection, basic commands, and list handling, have been optimized to compensate for reflective method invocations. Across all test cases, the new version performs better than the Frag 0.6 version. Tracing the execution time of reflective method calls revealed further hot spots in Frag, e.g., repetitive actions in list handling, which caused unnecessary garbage collection loads. To sum up, our design research project has also led to an improved and stable performance profile of the overall Frag implementation.

6. Related Work

In Section 2, we gave an overview of related work on guiding the adoption of a certain implementation technique for cross-language method invocations [5, 7, 9, 11, 23, 24, 27, 29, 33, 34]. It became clear that the findings offered turn out insufficient for making design decisions in a language development project such as Frag [13, 35]. The empirical evaluation approach presented in this paper offers an important analytical instrument applicable in various scenarios of cross-language integration. Besides, our experience report falls into the areas of designing embedded, dynamic languages in Java, in particular realizing forms of language symbiosis [15], and the implementation of reflection protocols in virtual machines such as current JVMs.

Designing and implementing symbiotic associations among two OO languages vary in several properties [11, 15]: features covered by the symbiotic associations realized; forms of (a)symmetry along the associations; the transparency of navigating such an association; the coverage of base-level and meta-object protocol elements; and, most interestingly, the extent to which the base- and meta-levels of the embedded and the host language overlap. The notion of *split objects* [34] helps bridge between the conceptual models of OO language symbiosis and guiding the actual design and implementation process. A split object has separate in-memory object representations in each of the two languages integrated (i.e., a wrapper and a wrappee half) while being treated as having a single, cross-language object identity. Mutual access to the split, yet shared state and behaviour is enforced through invocation forwarding between the respective wrapper and wrappee halves. Split object frameworks [34] further provide auxiliaries for converting back and forth between primitive and object-types as well as for mediating between parallel inheritance hierarchies. To apply the split object approach for the problem addressed in this paper, a cross-language invocation forwarder must be implemented (e.g., a two-way invoke method for the split object). This invocation forwarder negotiates signature information between the two split object halves (i.e., the method names, the invocation parameters, and possible return types).

An early example of an embedded language design based on split objects implemented in Java is the prototype-based Agora98 language [8]. Agora98 was not only one of the first languages to be hosted by Java (1.1, at that time), it also delivered a form of cross-language reflection and showcased the alignment of a prototypical and a class-centric object model. The Java implementation of the Piccola language, i.e., JPiccola [29], is a glueing language to facilitate composition of Java components. JPiccola embodies a language model following a process calculus and agents. The JPiccola interfaces to its Java host establish asymmetrically symbiotic associations between JPiccola and Java. Meta-level symbiosis is not covered, at the base-level, two-way associations are provided (e.g., through the javaObject and javaClass instructions). Frag [13, 35] itself continues a line of Java-implemented Tcl derivatives, namely Hecl [16] and Jacl [19], which both are limited to basic and asymmetric forms of symbiosis with Java.

Today, a variety of dynamic scripting languages and domain-specific languages (DSLs) built on top of host languages such as Java is available. A selection of embedded languages are Jython (built on top of the JVM; see [5, 22]), IronPython (built on top of the .NET CLR; see [18]), JRuby (built on top of the JVM; see [21]), and Rhino (built on top of the JVM; [26]). Another branch of embedded languages provides means to bypass their host languages and to directly access the latter's runtime environment (e.g., the VM) or even the machine instruction sets (see., e.g., R [25]).

Various infrastructures and toolkits for developing embedded (and, in particular, dynamic) languages have been proposed. While these frameworks embody design decisions for various forms of cross-language integration (i.e., method invocation, object creation, insulation methods, and inheritance; [11]), they also provide support for realizing dynamic language features on top of predominantly static host languages as well as for bridging two object-type systems. Higher-level instruments are host language extensions. Dynamic [5], for instance, provides a framework for embedded languages by emulating dynamic language features (e.g., method delegation, structural polymorphism, mixin-based inheritance) in the host language. This emulation is achieved by devising both introspection through the Java Reflection API and, based on the introspective input, runtime injection of bytecode to realize the adaptation and glue structures (e.g., classes) on demand.

Designing virtual machines as carrier platforms for multiple and heterogeneous (i.e., static vs. dynamic type systems) languages as been explored from various angles. For instance, embedded, yet intermediate languages for language-oriented programming (e.g., RPython [1], Slang [30]) have been considered which mediate between the requirements of realizing dynamic language features and type-aware virtual machines. Restricted Python (RPython; [1]), for instance, provides a language middleware on top of backends for the JVM and the .NET CLI. RPyhton balances the tensions of realizing dynamic language features in virtual machines, with virtual machines having typeful requirements built into their bytecode engines (e.g., mandatory type annotations for method dispatch targets as well as argument and return values). RPython does so by offering a mere subset of Python used as implementation vehicle for a feature-complete, self-hosted Python interpreter (i.e., PyPy). The .NET Dynamic Language Runtime (.NET DLR) takes a different approach, building upon an implicit, shared type system (i.e., by using type inference). As Frag [13, 35] is mono-typed with weak runtime checking of object-types, aligning an embedded and the Java object-type system was not an issue.

When considering and evaluating forms of reflective integration, an intimate knowledge about the reflection implementation in the runtime execution environment of the host language is paramount. For instance, a critical evaluation must consider the fact that reflection is often used internally in host language implementations to realize basic language features (e.g., for object serialization [4, 27]) or to support code generation [5]. Another, often ignored aspect is the share of reification in the total reflection overhead encountered [6]. Against the background of the Jikes Research Virtual Machine, Rogers et al. compare variants of indirection (i.e., "out-of-line" branching to the physical method memory locations) and bytecode generation to realize a reflective method dispatch [27]. They look at the time and space overheads linked to the creation of runtime representations of the method (and wrapper elements; i.e., the reification requirements), argument processing (parameter boxing), and the actual dispatch performance. They conclude that the acquisition of redirecting bytecode at the time of creating the method representation (i.e., the lookup operation and the eager creation of a reflective wrapper) creates the smallest overhead in execution time. The Frag performance data was collected running the stock Sun (OpenJDK) JVM, which implements the reflective object creation and the reflective method dispatch based on bytecode injection.

7. Lessons Learnt

In this design science research project [17], we demonstrated that a test suite covering critical aspects of a language system provides realistic insights for deciding on using reflective or non-reflective variants of cross-language method invocations. Using our novel idea to use transformation counts on cross-language refactorings, we were also able to quantify the positive impact of the reflective approach for rapid prototyping purposes. Transformation counts also highlight the potential of generative techniques and code-generating tooling for facilitating non-reflective integration strategies, especially when being compared to hand-written implementations of, for instance, switch-threaded wrappers.

Even in a project heavily relying on cross-language integration, such as Frag [13], the average performance impact of the reflective integration is not greater than 12 %. We identified some hot spots in our implementation, but a follow-up investigation allowed to fix these. With this, a more adequate estimate is that the average performance impact settles between 5-10 %. Unfortunately, it is not always feasible to realize a language feature on top of all implementation variants under comparison. For example, it would be interesting to incorporate the current, largely optimized Frag version 0.8 into the performance comparison. However, as features other than the implementation of cross-language method invocations have changed in the new version, the profiling results could not be attributed to the functional concern under review. Besides, the effort required to provide an additional Frag implementation, which incorporates yet another implementation variant, is considerable regarding the size and the complexity of the Frag core and test code bases (approx. 10,000 SLOC each). Nevertheless, our evaluation approach might generate valuable insights by just applying it on selected parts of the application and the corresponding test cases.

The transformation counts on cross-language refactorings have proven to be very helpful for quantifying the benefits of the implementation variants of cross-language method invocations. We found that, in average, the reflective variant considered entailed 71 % fewer transformations in four frequent cross-language refactorings (e.g., REMOVE HOST METHOD; see also Table 1). In scenarios which combine several cross-language refactorings (e.g., MOVE METHOD) the reduction in refactoring complexity is even higher. This leads us to the conclusion that the reflective integration techniques should be considered, provided that a modest degradation in execution times can be tolerated and refactoring efforts are critical. A thorough execution time analysis based on realistic test suites delivers valuable findings and provides guidance for perfective refactorings in an embedded language implementation. The figures from our project can be used for rough projections onto related development projects (e.g., in object remoting). Of course, for an in-depth analysis, project-specific evaluations must be conducted.

Providing two complete implementations of the Frag language core as complex benchmark applications, adopting different implementation variants of cross-language method invocations, is one major contribution of our work. The other one is the performance profile data generated from benchmarking these two language implementations. However, as already hinted at, the complexity of Frag impedes the inclusion of further implementation variants of cross-language method invocations throughout the entire language core. The two implementation techniques selected, i.e., a switch-threaded, non-reflective and a method-reflective variant (see Section 3 for details), settle at the two ends of the spectrum of implementation options available.

To give an example, we considered cross-language integration using dedicated language wrappers. Using wrappers is not strictly necessary in reflection-based integration [29]. Methods of the host language could be directly exposed to the embedded language, with the invocation data (parameters and return values) being automatically wrapped and unwrapped when crossing the language barrier. Dedicated method wrappers, however, permit us to implement different types of message passing control [3, 10] for Frag-to-Java method invocations. Also, wrappers allow to design signature interfaces of embedded objects independently from their host language implementation [29]. Furthermore, wrappers are the spots for applying optimizations (e.g., specific type conversion schemes, different embedded value representations, aligned I/O operations; [9]). In short, wrappers are particularly important when providing a development infrastructure for embedded DSLs such as Frag [13].

As for non-reflective integration, an implementation alternative to the switch-threaded wrapper evaluated for Frag are kinds of method objects. That is, each Frag/Java method could be represented as either a freestanding Java class or a class nested in the Frag/Java class. These would be Java-specific implementations of a COMMAND OBJECT [14] pattern variant applied to cross-language method invocations. We tentatively realized nested Frag/Java method objects by means of *anonymous inner classes* [31]. This way we learnt that method objects potentially combine some of the advantages of the non-reflective and the reflective approaches. First, they substitute the reflective reification and the reflective dispatch for an additional, delegating method call to the nested object. Still, method objects preserve some benefits of method-reflective Frag/Java methods (e.g., transparent access to variables of the parent object). Second, the Frag/Java methods are fully reified, though not by reflective reification. This preserves their composability (e.g., re-registering them under different names to the Frag interpreter). Third, the refactoring effort closely resembles the one reported for the method-reflective implementation because many switch-related transformation steps are not required anymore. A notable downside is that refining Frag/Java methods along a Java class hierarchy becomes impossible.

As for the method-reflective integration, certain optimizations based on the Java Reflection API may be considered to improve over the execution time degradation reported. In the Java context, for instance, one might consider adopting reflective invocation on per-class (i.e., *static*) methods, which incur a reduced execution time penalty. Applied to Frag, however, this would considerably increase the refactoring workload. This is because of the need for dedicated reflection proxies, either in terms of static methods within each Frag/Java class or additional proxy classes owning the static method forwarders.

As an example integrating reflective and non-reflective techniques, Cazzola [6] presents an approach to emulate native reflective method invocations in Java by generated proxies specific to and associated with each application class, borrowing the idea of *Java RMI stubs*. Each proxy provides a central dispatcher method `invoke`, modelled after the `java.lang.reflect.Method` signature interface. The dispatcher method implements switch-threaded method delegations, based on the target methods' hashed names as switch indices, with each switch branch early-binding a single, concrete target method. This scheme trades an improvement over the execution time degradation of standard reflective calls (i.e., 25 % in the evaluation reported in [6]) for an increased refactoring effort due to supplemental proxy classes and a switch control block.

More recently, a revised JVM infrastructure for managing dynamic method invocations has been introduced to the OpenJDK 7 [28]. Based on a new invocation bytecode (`invokedynamic`), allowing for deferring the binding to a particular object-type and receiver object, this infrastructure provides means to dynamically create and manage reifications of call sites and method handles. The reified call sites can be linked to different target methods at various decision points and times. Also, object representations of call sites and method handles are lazily acquired upon the first invocation. This acquisition step (i.e., the bootstrapping phase) can be intercepted from within the application (i.e., through the per-class bootstrap callbacks) and so allows for injecting application-specific call site and method handle types. When implementing cross-language method invocations, the language developer can therefore express language-specific linking points, linking times, method lookup, method dispatch, and parameter processing semantics in terms of this infrastructure (in Frag, e.g., varargs and the unknown method dispatcher).

For Frag, adopting this infrastructure appears promising. While preserving the advantages of dedicated method wrappers (e.g., message passing control, DSL-specific optimizations), this infrastructure devises *lightweight* object representations of methods (i.e., `java.dyn.Method` which sets aside annotations and selected object-type meta-data). Equally important, it avoids the execution time overhead of the Java Reflection API. First, there is no need for generating and injecting bytecoded invokers anymore. Second, the recurring, caller-specific method access checks have been replaced by one-time checks when creating method handles. Third, parameter processing can be adjusted according to the needs of the embedded language (e.g., ranging from automatic or no conversion at all to custom signature adapters). Fourth, standard JVM runtime optimizations (*inlining*) apply to such dynamic method invocations [28].

8. Conclusion

In this paper, we compared non-reflective and reflective implementation options for realizing cross-language method invocations between the embedded language Frag [13, 35] and Java as its host language. This comparison is performed in a qualitative and quantitative manner: On the one hand, cross-language method refactorings are evaluated with respect to the effort involved based on source transformation counts. On the other hand, an execution time comparison is presented. We found an average penalty in runtime performance of 12 % when adopting the Java Reflection API, which is justified in scenarios that do not have high performance requirements.

The general approach taken in this paper is applicable to many application scenarios of cross-language method refactorings. The actual evaluation results are, however, specific to the Java implementations as employed in Frag. As the Frag implementation uses a pretty standard reflection-based design and mostly basic Java library functionality, our results can be generalized to other dynamic language implementations in Java.

In a next step, we will collect further examples of cross-language refactorings by harvesting our ongoing DSL development projects based on Frag [13]. In addition, we plan to refine our performance measurements by incorporating ideas borrowed from benchmarking virtual machines [2]. This will include execution time measurements under different heap sizes and garbage collection schemes, the comparative reporting of multiple iterations to quantify setup costs and dynamic optimizations, and the micro-profiling of low-level hot spots (e.g., Frag's getter and setter methods).

References

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the DLS'07*, pages 53–54. ACM, 2007. URL `http://www.lst.inf.ethz.ch/research/publications/DLS_2007/DLS_2007.pdf`.

[2] S. M. Blackburn et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the OOPSLA'06*, pages 169–190, New York, NY, USA, 2006. ACM.

[3] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings of the ECOOP'98*, volume 1445 of *LNCS*, pages 396–417. Springer London, 1998.

[4] M. Braux and J. Noyé. Towards Partially Evaluating Reflection in Java. In *Proceedings of the PEPM'00*, volume 34 of *SIGPLAN Notices*, pages 2–11, 2000.

[5] T. M. Breuel. Implementing Dynamic Language Features in Java using Dynamic Code Generation. In *TOOLS39*. IEEE Computer Society, 2001.

[6] W. Cazzola. SmartReflection: Efficient Introspection in Java. *Journal of Object Technology*, 3(11):117–132, December 2004.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley Longman Publishing Co., Inc., 6th edition, 2000.

[8] W. De Meuter. Agora: The Story of the Simplest MOP in the World. In J. Noble, A. Taivalsaari, and I. Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*. Springer, Berlin/Heidelberg, Germany, January 1999.

[9] R. L. Drechsler and J. M. Mocenigo. The Yoix scripting language: a different way of writing Java applications. *Software – Practice and Experience*, 36:1–25, August 2006.

[10] S. Ducasse. Evaluating Message Passing Control Techniques in Smalltalk. *Journal of Object Orientated Programming*, 12:39–44, June 1999.

[11] T. Ekman, P. Mechlenborg, and U. P. Schultz. Flexible Language Interoperability. *Journal of Object Technology*, 6(8):95–116, 2007.

[12] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] Frag. `http://frag.sourceforge.net/`, 2009.

[14] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley, October 1994.

[15] K. Gybels, R. Wuyts, S. Ducasse, and M. D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Computer Languages, Systems & Structures*, 32(2-3):109–124, 2006.

[16] Hecl. `http://www.hecl.org/`, 2010.

[17] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.

[18] IronPython. `http://ironpython.net/`, 2010.

[19] Jacl. `http://tcljava.sourceforge.net/docs/website/index.html`, 2010.

[20] Janino. `http://www.janino.net/`, 2009.

[21] JRuby. `http://jruby.org/`, 2010.

[22] Jython. `http://www.jython.org/`, 2010.

[23] D. Kearns. Choosing a Java scripting language: Round two. JavaWorld.com, Online article: `http://www.javaworld.com/javaworld/jw-03-2005/jw-0314-scripting.html`, last accessed: January 21, 2010, March 2005.

[24] M. Kempf, R. Kleeb, M. Klenk, and P. Sommerlad. Cross Language Refactoring for Eclipse Plug-ins. In *Proceedings of the WRT 2008*. ACM, 2008.

[25] C. Müller and A. Lumsdaine. Runtime Synthesis of High-Performance Code from Scripting Languages. In *Proceedings of the OOPSLA'06*, pages 954–963. ACM, 2006.

[26] Rhino – JavaScript for Java. `http://www.mozilla.org/rhino/`, 2010.

[27] I. Rogers, J. Zhao, and I. Watson. Approaches to Reflective Method Invocation. In *Proceedings of the ICOOOLPS'08*, 2008. URL `http://icooolps.loria.fr/icooolps2008/Papers/ICOOOLPS2008_paper08_Rogers_Zhao_Watson_final.pdf`.

[28] J. R. Rose. Bytecodes meet Combinators: invokedynamic on the JVM. In *Proceedings of the VMIL'09*, New York, NY, USA, 2009. ACM.

[29] N. Schärli, F. Achermann, and O. Nierstrasz. Meta-level Language Bridging. Technical report, University of Berne, 2002. URL `http://scg.unibe.ch/archive/drafts/bridging.pdf`.

[30] Slang. `http://wiki.squeak.org/squeak/2267`, 2010.

[31] Sun. About Microsoft's "Delegates". `http://java.sun.com/docs/white/delegates.html`, 2010.

[32] SWIG. `http://www.swig.org/`, 2009.

[33] S. Vinoski. A Time for Reflection. *IEEE Internet Computing*, pages 86–89, January-February 2005.

[34] U. Zdun. Using Split Objects for Maintenance and Reengineering Tasks. In *Proceedings of the CSMR'04*, pages 105–114. IEEE Computer Society, 2001.

[35] U. Zdun. A DSL toolkit for deferring architectural decisions in DSL-based software design. *Information and Software Technology*, 52(7): 733–748, 2010.