Supporting Incremental and Experimental Software Evolution by Runtime Method Transformations

Uwe Zdun

New Media Lab, Department of Information Systems, Vienna University of Economics and BA, Austria

Abstract

Transformations of object-oriented methods are a prevalent object-oriented programming technique, but in many languages they are not supported at runtime. Therefore it can be hard to apply method transformations for incremental or experimental software evolution, or other problems that require runtime software behavior adaptation. The goal of the work presented in this paper is to provide a better conceptual and technical support for runtime method transformations. A non-intrusive model for method transformations and a set of runtime method transformation primitives are presented. We also present a pattern language for implementing dynamic method abstractions and combining them with languages that do not support dynamic methods natively. As a case study we introduce a runtime transformation framework for the dynamic configuration and composition language Frag, its connection to Java, and an end user programming example.

Key words: Runtime Method Transformation, Software Evolution, Software Adaptation, Patterns

Email address: `zdun@acm.org` (Uwe Zdun).

1 Introduction

Runtime software adaptation and evolution is required in many programming situations. The standard solution, supported by most object-oriented languages, is the association (or delegation) relationship. By changing the association or delegation link, the program behavior can be modified at runtime. In many programming situations, especially in the presence of *unanticipated software evolution* [1], developers would benefit from a more sophisticated runtime adaptation and evolution support. In this paper, we aim at supporting situations in which it is necessary to specify a new behavior for a program at runtime and the necessary changes cannot be anticipated before runtime. Such requirements occur frequently when a program needs to be changed incrementally, or runtime experimentation is required for the programming task.

Consider a typical example situation: many activities of reengineering a software system are rather experimental or incremental in their nature. Understanding a given legacy system, for instance, often means to "play" with the existing system, add traces, modify the source code, etc. Wrapping a legacy system means to create an initial set of wrappers first and then evolve these wrappers incrementally. Logging and tracing a system's call structures often means incrementally adding traces until the relevant control flows are captured. In summary, many typical reengineering activities have an experimental or incremental nature *and* are dependent on runtime semantics. Having to recompile the system for each change can rather hinder experiments or incremental evolution.

Situations, similar to the reengineering example, arise in many other domains. Just consider two other examples (from projects the author was involved in). Evolution in scientific software can be supported by letting the domain expert experiment with the implementation of the scientific algorithms while the software is running [2]. Similarly, content editors of digital television applications benefit from being able to incrementally change the applications while they are running to see the effects of changes directly and to foster rapid application development (see Section 4.4 for an example from this domain).

Many approaches are proposed to cope with adaptation and evolution of object-oriented programs. Runtime adaptations are an integral part of a number of more dynamic, object-oriented environments, such as CLOS [3], Smalltalk [4], and Self [5]. These languages provide both a programming environment and a program execution environment, allowing one to influence the language behavior from within a program. Different language constructs, such as computational reflection [6,7], Lisp macros [8], meta-object protocols (MOP) [9], meta-classes, dynamic method lookup and dispatch, and dynamic classes are supported for this purpose. These constructs provide a great power to the programmer; yet they also pose a high complexity: in order to understand some expression in the language, the current runtime definition of the environment has to be understood.

Similar dynamic and introspective languages features are provided by a number of scripting languages, including Tcl [10], Python [11], Perl [12], and Ruby [13]. These languages provide these features in a more "tamed" way because they follow a different approach to system development than system languages [14]: instead of developing the whole system in only one language, a two-language approach is chosen. Reusable components are written in system languages, such as C, C++, or Java, and the scripting language is used as a glue to compose and configure these components. For other tasks, such as incremental or experimental software evolution, the dynamic and introspective languages features are mostly used in the course of daily programming and not as a distinct adaptation or evolution technique.

A number of approaches have been proposed to support object-oriented evolution and adaptation without a need for meta-programming or reflection. Instead (static) program transformation is used. Examples are aspect-oriented programming (AOP) [15,16,17,18] and refactoring [19,20]. Program transformation generally refers to techniques for automating programming tasks to increase the programmer's productivity [21]. There are many other application areas for program transformation in software engineering, including compilation, optimization, refactoring, software renovation, and reengineering. These (and many similar) techniques have in common that they are easier to understand and apply for the average programmer than meta-programming or reflection. Yet they are hard to apply in the context of runtime evolution and adaptation because they focus on static adaptation techniques.

Aspect-oriented adaptation constructs can also be composed at runtime. For instance, there are a number of AOP approaches that offer runtime aspect composition [22,23,24]. Dynamic message interceptors in programming languages (such as XOTcl filters and mixin classes [25]) or message interceptors in popular middleware (e.g. [26,27]) can be used to implement runtime composition of aspects as well (see [28]). These approaches require either the system to be statically instrumented before the runtime adaptation takes place, or the programming language or middleware must support runtime adaptation constructs.

Another important issues is that scenarios of incremental or experimental software evolution often require a simple interface for the adaptations. For example, for the reengineers, scientists, or content editors in the examples given above, it is important that the offered programming interfaces are simple and specific to the work task [29]. However, in virtually all adaptation approaches discussed above, not only the specific work task implementation has to be understood, but also the system and its environment, such as the system structures to which the adaptation is applied, the meta-object protocol, the reflection system, the aspect language, or the adaptation constructs.

This paper proposes runtime method transformations to support incremental or experimental software evolution. Runtime method transformations are very similar in their application to (static) program transformations. Yet, internally they are implemented using a reflective object or class system. This system is completely hidden from the developer, so that it does not add further complexity. Within the local context of the transformed methods, the full power of a programming language is available. As we will show, runtime method transformations combine three important characteristics: they can be used for runtime adaptation, they introduce no new, complex abstractions but use the simple, familiar method abstraction, and they can be applied locally in the context of a particular work task. All three characteristics cannot be found together in any of the approaches discussed above.

As a foundation of runtime method transformation, we provide a conceptual framework and terminology for runtime adaptation (presented in Section 2). Whereas working with runtime method transformation is non-complex and simple, the design and implementation of a runtime method transformation framework is a non-trivial task in most programming languages. To support the design and implementation of runtime method transformation frameworks following the concepts from Section 2, a pattern language is provided in Section 3.

We do not only investigate runtime method transformation at a conceptual level, but we also describe a prototype implementation: the Tcl extension Frag [30] (see Section 4). Frag is an object-oriented Tcl [10] extension that is designed to be used for configuration and composition tasks. Frag supports a reflective object and class system and can be combined with other languages (currently C, C++, Tcl, and Java). To the best of our knowledge, none of the aspect and adaptation frameworks discussed above supports such language diversity. On top of this infrastructure, we present a simple trace example and a more complex end user programming example from the area of interactive game scripting.

2 Runtime Method Transformations

A *method transformation* is a kind of program transformation, and it can be defined as any possible change of the definition of an (object-oriented) method. A method definition comprises a method name, a method scope (usually a specific class) in which the method is defined, method parameters (possibly also parameter types and parameter order), a return type, and a method body. All these elements can possibly be affected by a method transformation. As we change these elements frequently during ordinary object-oriented programming practices, method transformations are a prevalent technique of object-oriented programming.

A *runtime method transformation* is a method transformation that is applied to a method while the system runs. Note that the notion of dynamic methods is nothing new. As we discuss in Section 3, dynamic methods are a common pattern in Lisp variants, Smalltalk, many popular scripting languages, and even in Java [31]. As pointed out in [31]: "dynamic methods are at the same time a powerful and a dangerous device. When used properly, they offer unique possibilities to extend and retroactively modify software systems. On the other hand, when used inappropriately, they make it quite easy to cause havoc by overriding dynamic methods in a completely nonsensical way." The goal of this work is to provide a non-intrusive, limited model with which dynamic methods can be applied safely as a dedicated software engineering approach.

2.1 A Non-Intrusive Method Transformation Model

A method can have an initial definition that is altered by a runtime method transformation. There are different goals of our approach that require some kind of non-intrusiveness of method transformations:

- The runtime method transformations should be usable as a dynamic adaptation technique. That is, we use it to add decorations or adaptations to existing methods, such as trace code. We should also be able to dynamically remove these decorations or adaptations again when no longer needed. Intrusive method transformation would make it hard to remove once added code.
- Experimentation and incremental evolution with method transformations should be supported. This goal implies that once made, additions should be (easily) removable. We should also be able to distinguish different additions, if there are more than one, because any of them might be removed.
- A goal of our work is tool support for runtime method transformations. Besides applying the individual method transformations, the tool can highlight different transformations or additions, for instance, in different colors. To do so, the tool needs to find out what is an addition and what belongs to the original method definition.

Note that non-intrusiveness is only a goal for some kinds of method transformations, others can be inherently intrusive in nature. Once developers are sure that some method transformations are mature, it should be possible to take these over into the productive system. To resolve these forces we propose to distinguish the following concepts:

- *Method Intrinsics:* We call the original or inherent definition of a method the intrinsics of the method (according to the intrinsic object definition in role concepts [32]). There are some runtime method transformation primitives that change intrinsics of the method. When some additions become mature, developers can decide to migrate these additions into the intrinsics of the method. We distinguish the intrinsic signature (return type and parameter list) and the intrinsic body. Note that a change of one of them does not necessarily imply a change of the other. A change of the intrinsic signature often implies some interface incompatibility.
- *Method Extrinsics:* Extrinsics are results of runtime method transformations that are stored separately and do not change the intrinsics of the method. The advantage of method transformations that do not alter the intrinsics is that additions can also be deleted automatically after they have been added. When working with a tool, a second advantage is that we can highlight the additions. We distinguish before-code, after-code, and extrinsic signature elements (return type and parameter list).
- *Current Method Definition:* There is always a current method definition, which is actually executed, when the method is called. This is a runtime representation of the method. The current method definition is a composition of method intrinsics and method extrinsics. Initially the current method definition equals the method intrinsics, but it changes when runtime method transformations are applied.
- *Method Definition in the Program Text:* The current method definition is a runtime representation of the method that has to be distinguished from the method representation in the program text. Especially for tools it is important to have a means to write a changed method back into the system's program text. In other words, the environment should support some means to serialize the current method definition at runtime.

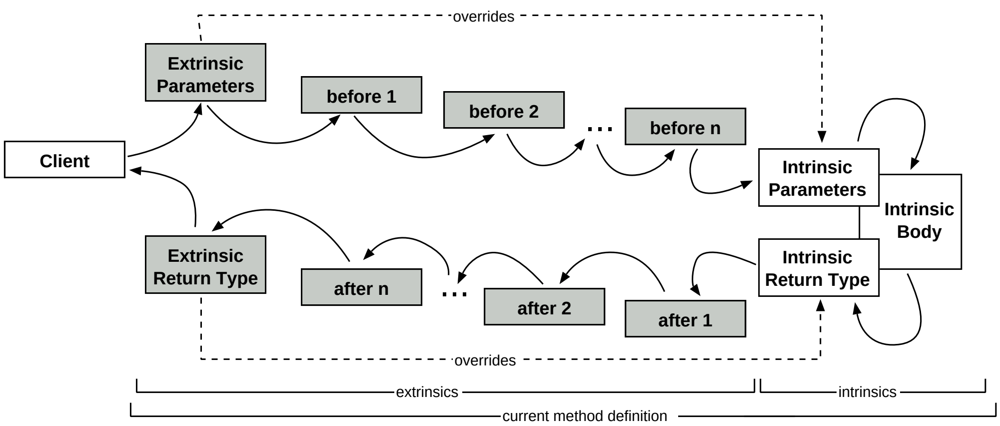

Figure 1. Construction of the Current Method From Extrinsics and Intrinsics

These elements of our method transformation model define how a method is constructed in the presence of runtime method transformations (see Figure 1). These model elements imply also the possible activities and transitions, performed by the different kinds of method transformations (depicted in Figure 2). Before any method transformation can be applied, the original method definition has to be stored as intrinsics. Next we can apply method transformations that change either the intrinsics or the extrinsics. An intrinsic change directly affects the stored intrinsics, whereas an extrinsic change is non-intrusive. There is a special activity "make current method intrinsic" to make the current method the intrinsic method definition; that is, we accept any extrinsic change performed so far. This activity is typically implemented by serializing all extrinsic and intrinsic elements into one method text, which is then used to override the original intrinsic method definition. After performing a number of method transformations during a system run, the changes can either be discarded or written back into the program text.

Note that the "discard" path is not only used when a method transformation experiment has failed, but also when using runtime method transformations as a programming technique. Consider, for instance, a set of methods of a component C is adapted with a trace aspect. We only want to change the current runtime method definition for as long as the system runs, but not the method definition in the program text. Otherwise other applications using the component C would have to use the trace aspect as well, what is not intended. Thus, the changes are only applied for one application, and when the application stops, the changes are discarded.

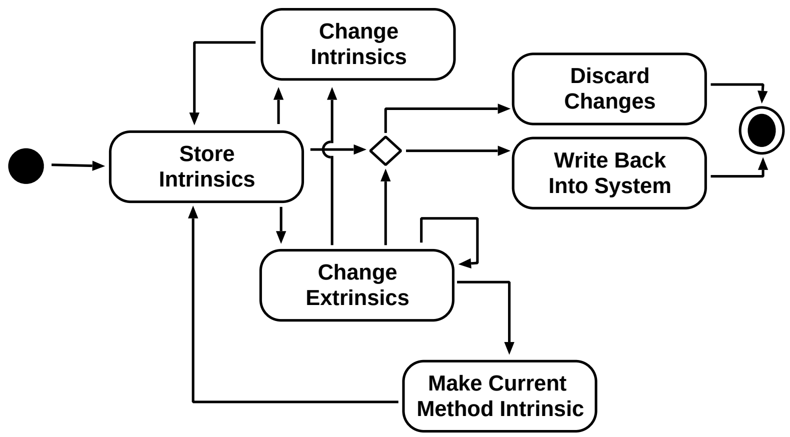

Figure 2. Activities and Transitions in the Method Transformation Model

2.2 Runtime Method Transformation Primitives

In this section, we introduce the set of runtime method transformation primitives as provided by our transformation framework, introduced in Section 4.

Of course, this is not a complete set of all possible runtime method transformation primitives, but it is sufficient for practical work with runtime method transformations. Some of these primitives are also covered (in part) by the refactorings documented in [20], what we indicate accordingly.

Some runtime method transformation are applied only in the scope of a single method. Other runtime method transformations are externally visible; that is, they either affect the signature of the method or even classes and class hierarchies. As discussed in the previous section, we can further distinguish changes to the method intrinsics and non-intrusive changes to the method extrinsics. When using these criteria for discriminating runtime method transformations, we can derive four categories that we describe in more detail in the remainder of this section (see also Figure 3).

There is a set of primitives that belong to the *class/hierarchy change* category. All primitives in this category affect the method's class or class definition when the method transformation is applied. These transformations alter the method intrinsics. Typical primitives in this category are:

- *Add Method:* A new method is added to a specified class.
- *Delete Method:* A method is removed from a specified class.
- *Copy Method:* The method is copied to another destination, given by a class and method identifier. Copy Method can be implemented as Add Method at target.
- *Move Method:* The method is moved to another destination, given by a class and method identifier. This primitive might break client code. If only the method identifier changes, the primitive can be used to rename a method. Move with rename can be made compatible by Substitutions on all affected client code. The transformation covers the refactorings Rename Method and Move Method from [20]. Move Method can be implemented as Copy Method followed by Delete Method of the source method.
- *Pull Up Method:* This primitive is a kind of Move Method with a superclass as destination. It should only be applied, if it makes sense to apply the method for all subclasses of the destination class. There is a same-named refactoring in [20]. Pull Up Method can be implemented as Add Method (at superclass) and Delete Method (from subclass).
- *Push Down Method:* This primitive is a kind of repeated Move Method with a list of subclasses as destination. It can be applied to all subclasses or to any subset of them. There is a same-named refactoring in [20]. Push Down Method can be implemented as Add Method (at subclasses) and Delete Method (from superclass).

Another set of primitives belong to the *before/after-code change* category. All primitives in this category affect only the method's extrinsics and are method internal changes. Thus these primitives are mainly used for decoration and

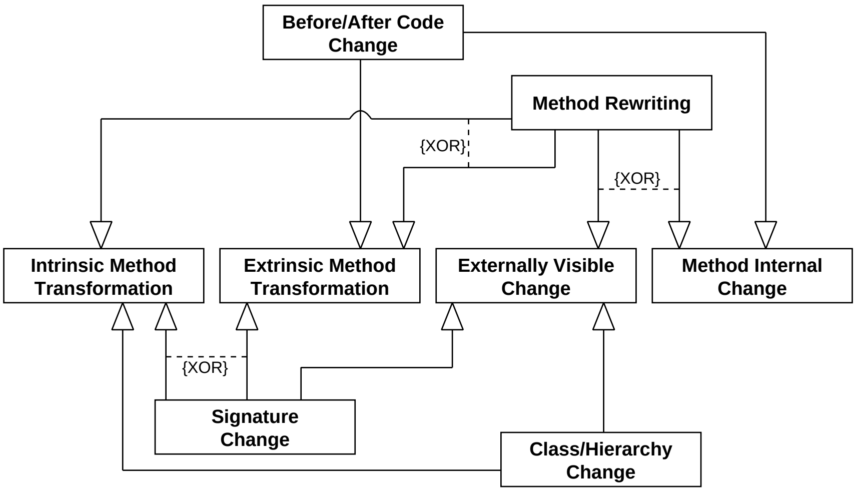

Figure 3. Categories of Runtime Method Transformation Primitives

adaptation tasks:

- *Add Code Before Method Body:* This primitive adds a given code snippet before the method body.
- *Delete Code Before Method Body:* This primitive deletes a specific piece of "before-code" that has been added before.
- *Delete All Code Before Method Body:* This primitive deletes all "before-code" that has been added.
- *Add Code After Method Body:* This primitive adds a given code snippet after the method body.
- *Delete Code After Method Body:* This primitive deletes a specific piece of "after-code" that has been added.
- *Delete All Code After Method Body:* This primitive deletes all "after-code" that has been added.

Aspect-oriented systems also define the "around" category, meaning that some code is executed instead of an original method definition. As an intrinsic change, around code can be implemented using Delete Method of the original method definition followed by Add Method of the new definition. This change would overwrite the method definition. Sometimes forwarding to the original behavior from the around code is required. This can be done using Move Method instead of Delete Method. Then the new method can refer to the moved method. Transparent, extrinsic around behavior is more difficult to achieve using method transformations: we additionally need an automatic forwarding mechanism. As an example solution, we will introduce Frag's mixin classes in Section 4. An Add Method on a mixin class provides an extrinsic around implementation that can automatically forward invocations using Frag's `next` primitive.

The primitives in the category *signature change* affect either the intrinsic *or* extrinsic signature of a method. Thus they are externally visible transformations:

- *Change Parameters:* This primitive exchanges the (extrinsic or intrinsic) parameter list of a method. The two refactorings Add Parameter and Remove Parameter also change the method parameters [20]. Note that for positional arguments (as in most programming languages) this transformation might break client code, except if it is possible to provide default values (as for instance in Tcl). Non-positional arguments, as in SOAP [33], can also avoid this problem. Another solution is to first apply Copy Method, and then change the parameters on the copy only, what can be applied if such polymorphism is support by the language.
- *Change Return Type:* This primitive exchanges the (extrinsic or intrinsic) return type of a method. It changes the method argument intrinsics and potentially changes the method signature. Note that this transformation might break client code. Again, Copy Method followed by Change Return Type can solve this problem. Some languages, such as Tcl, have only one generic return type (in Tcl: strings), making this primitive obsolete.

The primitives in the category *method rewriting* apply a specified substitution for the method and rewrite it accordingly:

- *Substitution:* There are many possible ways to specify substitutions. We use regular expressions in our work for this purpose, of course, other substitution or rewrite rules are also possible. Furthermore, one can also specify to which parts of the method the substitution should be applied: the method body intrinsics, the signature intrinsics, the signature extrinsics, the before-code, or the after-code. Any combination is also possible. The primitive thus can possibly change the method intrinsics or the extrinsics. The primitive can be applied as a method internal change only, or be externally visible, say, by changing the method's intrinsic signature.
- *Exchange Body:* The intrinsic body of a method is exchanged with another body, which is specified. This primitive is similar to the refactoring Substitute Algorithm [20].
- *Make Current Method Intrinsics:* This is a special substitution that accepts the current before-code and after-code as intrinsics and then applies the Delete All Before-Code and Delete All After-Code primitives. Note that this primitive is a part of our activities model (see Figure 2) and is required for implementing the model fully.

On top of the transformation primitives defined before, we can define complex method transformations, as for instance the following examples:

- *Extract Method:* A new method is added with Add Method. Its body is defined as a part of another method. The code in this other method is exchanged by Substitution with an invocation to the new method. There is a same named refactoring [20].

- *Inline Method:* A method is deleted with Delete Method. The body of the method is inlined in all client methods by Substitution. Note that the parameters have to be adapted or substituted as well. There is a same named refactoring [20].
- *Model-Based Rewriting:* The substitutions in the "method rewriting" category are basic primitives that are sufficient for experimental changes and simple replacements. For some tasks it might be beneficial if the substitution recognizes model elements in the text to be substituted. Then rewrite rules such as "rename all occurrences of the class X in a given code snippet" can be written. We have not implemented such support in our transformation framework yet, but it can be done using the pattern INTROSPECTION OPTIONS (see Section 3).

3 Design and Implementation of a Runtime Method Transformation Framework: A Pattern Language

In the preceding sections we have assumed that the used language provides support for dynamic methods. However, most mainstream languages, such as C, C++, or Java, do not provide such a language construct. In this section, we present a pattern language as a conceptual, language-independent foundation for designing and implementing a technical infrastructure for runtime method transformations.

A *pattern* is a recurring solution to a problem in a context, resolving a set of forces. Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution [34]. A *pattern language* is a collection of patterns that solve the prevalent problems in a particular domain and context, and, as a language of patterns, it especially focuses on the pattern relationships in this domain and context. As an element of language, a pattern is an instruction, which can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant [34].

The patterns, described in this section, are sufficient to design and implement a runtime method transformation framework and combine it with a given host language, even if that language does not support dynamic method abstractions. We have implemented runtime method transformation frameworks within the implementation of XOTcl [25], an object-oriented Tcl variant written in C. Also we have implemented a method transformation framework in Frag [30], a Tcl extension written in Tcl itself (see Section 4).

Figure 4 shows an overview of the patterns in the pattern language and their most important relationships. This pattern language has a strong relation to two 'external' patterns from [35]. A COMMAND [35] encapsulates an invocation to an object and provides a generic (abstract) invocation interface. COMMANDS alone only allow for adaptation by changing the association link to a COMMAND. In the pattern language the COMMAND abstraction is extended with additional indirections to allow for building and interpreting dynamic method abstractions. The important interpretation step is implemented using another pattern from [35], the INTERPRETER pattern. In general an INTERPRETER defines a representation for a grammar along with an interpretation mechanism to interpret the language. These two patterns are used within some of the other patterns of the pattern language. The other patterns are in particular:

- A DYNAMIC METHOD provides a method abstraction that can be modified, added to a class, and deleted from its class at runtime.
- A COMMAND LANGUAGE [28] provides a symbolic (e.g. string-based) language which is mapped to COMMANDS using an INTERPRETER.
- A METHOD COMMAND is a special COMMAND used for defining (and redefining) methods at runtime. Thus it can be used to implement DYNAMIC METHODS in languages that provide no suitable abstraction natively.
- An INTROSPECTION OPTION [36] provides introspection of software structures and dependencies, defined for instance within an INTERPRETER.
- A CALLSTACK contains one callframe for each invocation (for instance within an INTERPRETER). It is used to maintain per-call data.
- An INVOCATION CONTEXT [36] describes the current invocation and provides access to per-call data for COMMAND LANGUAGE objects.
- A SPLIT OBJECT is an object defined half in a host language and half in the COMMAND LANGUAGE. This way host language objects can be accessed from within the COMMAND LANGUAGE, and vice versa.
- A HOOK INJECTOR [36] inserts invocations (hooks) into a given program (e.g. into the parse tree or byte code). It can be used to insert SPLIT OBJECT invocations into a given program.

3.1 Dynamic Method

Context Runtime modification of the system's behavior is required.

Problem Consider a situation in which modifying the system's behavior at compile time, binding time, or load time is too early. For instance, a reengineer who wants to add traces to specific parts of a system would benefit from adding these changes at runtime while working with a reengineering tool. In this scenario, a hands-on, runtime approach is required for experimentation. The approach should also provide a conceptual framework so that the "final" changes (after experimenting a while) can be incorporated into the system.

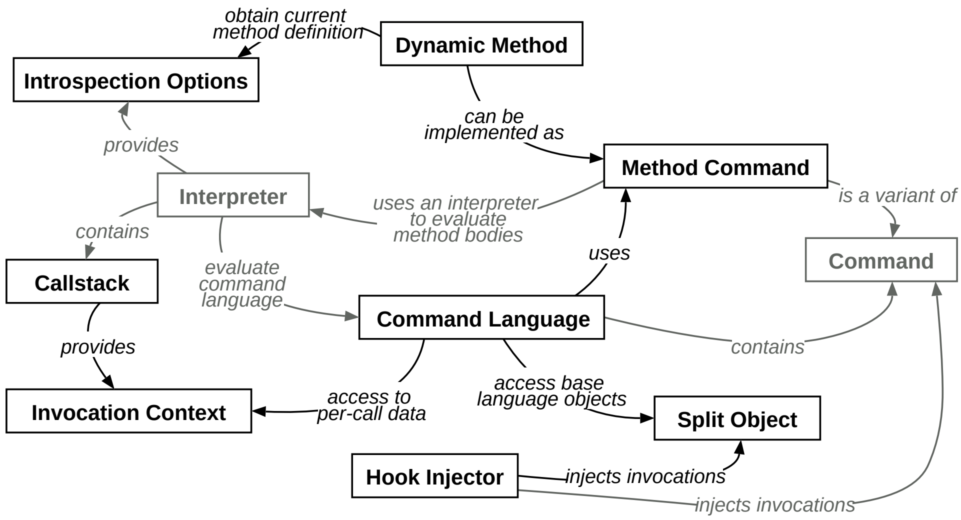

Figure 4. Pattern Language Overview

The quality of the code should not suffer from this kind of software evolution.

Solution Provide a DYNAMIC METHOD abstraction, so that method definitions can be added to a class and removed from a class (or object) at runtime. This way the method definition can be modified at runtime as well (by replacement). For instance, the method definition can consist of a number of strings containing the class name, method name, method parameters (and parameter types), return type, and method body.

Figure 5 shows an INTERPRETER that reads a script, defining a method `myMethod`. At runtime, a client first adds another method `log` and then redefines the method `myMethod` to use `log` for a trace output message. Alternatively, the client could also remove methods from the running system.

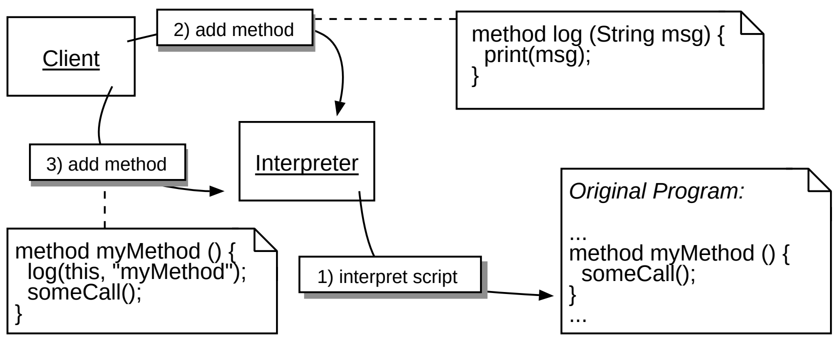

Figure 5. Method definition and redefinition at runtime

Discussion DYNAMIC METHODS highly raise the runtime flexibility of a system and enable dynamic (i.e. interactive or experimental) system evolution. But without further conceptual support, they can add complexity to a system. That is, for understanding a system, a developer needs to know the current method definitions at any time. Note that the approach proposed in this paper provides such conceptual support.

With DYNAMIC METHODS only, it is hard to evolve a system incrementally because a developer cannot find out about the old method definition. Remembering the method definition locally does not help much, because possibly another part of the program has changed the method in the meantime. Changing the method would then discard these changes. INTROSPECTION OPTIONS for DYNAMIC METHODS provide a solution to this problem. They allow one to query the currently defined methods and method definitions at runtime.

A DYNAMIC METHOD can be implemented with a special kind of COMMAND [35], a METHOD COMMAND. In interpreted languages the DYNAMIC METHOD can also be implemented within the language's INTERPRETER [35].

DYNAMIC METHOD code might be inconsistent with the rest of the system, leading to runtime exceptions. For instance, a DYNAMIC METHOD'S code might refer to another method that does not exist. It is the responsibility of the client providing a new method definition to ensure consistency. INTROSPECTION OPTIONS enable the client to check the environment for consistency.

3.2 Command Language

Context Multiple COMMANDS are used within one system.

Problem Using many COMMANDS (say for configuring a system) without further support can be cumbersome in some in cases, where the COMMANDS have to be assembled in various different ways. Runtime composition of the COMMANDS is not possible if the composition is hard-coded into static, compiled languages (such as C, C++, or Java). The code of multiple, consecutive COMMAND invocations might be hard to read, just consider the following simple example:

```
if (exprCmd.execute()) {
  result = doCmd1.execute();
  doCmd2.value = result;
  doCmd2.execute();
}
```

Solution Express COMMAND composition in a COMMAND LANGUAGE, instead of calling the commands directly using an API. Each COMMAND is accessed with a unique command name. The host language, in which COMMANDS are implemented, embeds the COMMAND LANGUAGE. In the host language, the COMMAND LANGUAGE'S INTERPRETER or compiler is invoked at runtime to evaluate the COMMANDS expressed in the COMMAND LANGUAGE. Thus COM-

MANDS can be composed freely, even at runtime.

Almost all elements of a COMMAND LANGUAGE are COMMANDS, what means that a COMMAND LANGUAGE'S syntax and grammar rules are usually quite simple. There are some additional syntax elements, as for instance grouping of instructions in blocks, substitutions, and operators (examples are operators for assignments, ends of instructions, or expressions).

COMMAND LANGUAGE code is typically expressed as strings of the host language, in which the COMMAND LANGUAGE is implemented. From within the host language, code can be evaluated in the COMMAND LANGUAGE, and the results of these evaluations can be obtained.

Consider again the above simple example. Using a COMMAND LANGUAGE we can provide the dynamic expression by variable substitution (with '`$`') and pass the result of `doCmd1` as an argument to `doCmd2` (with '`[...]`'). These changes shorten the resulting code and make it much more readable:

```
if {$expr} {
  doCmd2 [doCmd1]
}
```

Discussion Often existing COMMAND LANGUAGES, such as scripting languages, can be reused. Thus, to use a COMMAND LANGUAGE usually does not mean that developers have to implement a full-fledged programming language from scratch.

In languages that support DYNAMIC METHODS, COMMAND LANGUAGE and host language can be identical. In other cases, we require some kind of language integration in order to use a host language object from within a COMMAND LANGUAGE, and vice versa. This integration is provided by SPLIT OBJECTS.

To avoid two different invocation styles in a compiled host language, we can use a HOOK INJECTOR to add COMMAND LANGUAGE invocations into host language code.

3.3 Method Command

Context DYNAMIC METHODS should be implemented.

Problem Many programming languages do not allow for changing method definitions at runtime, but this is a requirement for DYNAMIC METHODS. How can we implement a DYNAMIC METHOD abstraction in a language that does only supports methods that are compiled before runtime?

Solution Implement a method abstraction as a special COMMAND [35]. A

METHOD COMMAND is a COMMAND variant that evaluates the embedded method definition when the COMMAND is executed. It also adds two more functionalities to standard COMMANDS: first, it allows one to provide and change the method body and argument definitions at runtime. Secondly, it provides a connection to an INTERPRETER or on-the-fly compiler for the language in which the method bodies are written.

Figure 6 shows a client that invokes a DYNAMIC METHOD using an INTERPRETER. The INTERPRETER looks up the METHOD COMMAND corresponding to the provided method name in the COMMAND table of the class of the invoked object. The returned METHOD COMMAND object contains the DYNAMIC METHOD data (arguments and body) and can be executed within the INTERPRETER. The result of the invocation is returned to the client.

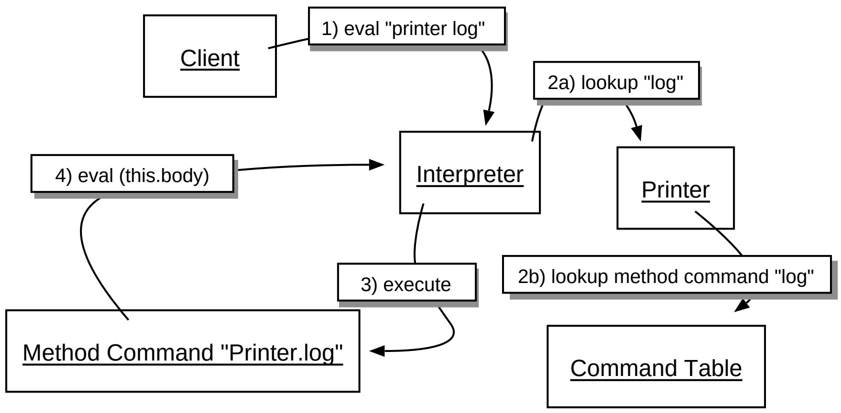

Figure 6. Invoking a dynamic method encapsulated in a method command

Discussion METHOD COMMANDS enable clients to generically define DYNAMIC METHODS, even if the programming language does not support them. As a drawback in such languages another style of invocation is required for the DYNAMIC METHODS. The client has to write something like:

```
interpreter.eval("printer log");
```

instead of for example:

```
printer.log();
```

A HOOK INJECTOR can inject such invocations and thus help to avoid this problem.

If on-the-fly compilation of DYNAMIC METHODS is supported, the METHOD COMMAND also maintains a compiled version of the method. Runtime (bytecode) compilation is usually performed lazily: after the method has changed, the compiled version is invalided and compiled again for the next use.

The pattern relies on the INTERPRETER [35] pattern. The INTERPRETER possibly can implement a very simple language (e.g. a domain-specific language), or a more complex one. One should consider to use an existing language's IN-

TERPRETER rather than writing a new language INTERPRETER from scratch. The main tasks of the INTERPRETER are parsing and interpreting the languages grammar, and mapping COMMAND invocations to implementations, including invocations of (other) METHOD COMMANDS.

3.4 Introspection Options

Context Information about the software structures and dependencies of a system is required at runtime.

Problem Many architectural structures and dependencies of a software system are needed while it runs. These structures and dependencies include dynamic structures (that can change at runtime) as well as static structures (that are defined at compile time and do not change at runtime). But in many programming languages there is no integrated and extensible way to obtain this information at runtime.

In the case of DYNAMIC METHODS, making sensible changes to a method definition often requires knowledge of the original behavior. Or, in other cases, the original behavior should be preserved in some way. That is, the original method definition of a DYNAMIC METHOD is required at runtime.

Solution Offer INTROSPECTION OPTIONS for each interesting architectural element (e.g. in the INTERPRETER). For instance, for DYNAMIC METHODS let developers obtain the original method definition by offering INTROSPECTION OPTIONS for the METHOD COMMANDS. Provide options for retrieving the method body (as a string containing the program text) and the argument list with argument names and argument types.

Figure 7 shows how INTROSPECTION OPTIONS can be used to access an INTERPRETER'S internal structures. Here, the COMMAND table of a particular class `C3` is queried for a list of the METHOD COMMANDS.

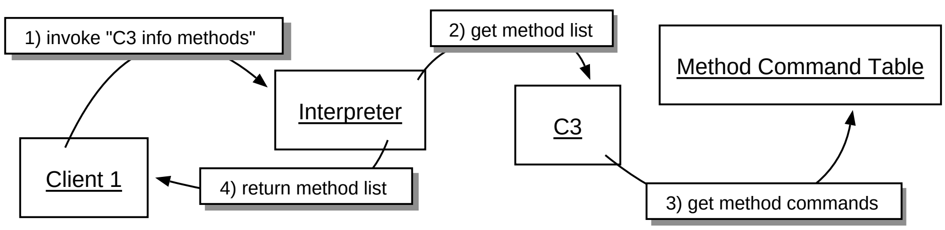

Figure 7. Querying the command table using an introspection option

Discussion When using INTROSPECTION OPTIONS and DYNAMIC METHODS, the consistency problem of remembering the current method definition for redefining methods can be avoided. The METHOD COMMAND itself remembers its definition and provides it using INTROSPECTION OPTIONS. Typical INTROSPECTION OPTIONS for methods are parameters, body, parameter types, and return type.

INTROSPECTION OPTIONS and DYNAMIC METHODS can be used to incrementally make changes to a method. The method definition can get more and more refined. An original behavior definition can then be used as an example. Thus incremental changes of a given implementation are possible.

3.5 Callstack

Context An INTERPRETER or another runtime dispatch mechanism is used.

Problem State should be preserved or manipulated at a given point in the control flow. This problems occurs, for instance, when implementing an INTERPRETER. The INTERPRETER needs to preserve the currently executing object, class, and method, when invocations take place from within the currently executing method.

Solution The control flow can be abstracted into a number of invocations. Let the INTERPRETER instantiate one callframe per invocation and push it onto a CALLSTACK. When the invocation (and all inner invocations) have finished, the callframe is popped from the CALLSTACK. The callframe contains all per-call information needed by the system.

Discussion Typically an INTERPRETER or another runtime dispatch mechanism requires some way to maintain per-call information. However, a CALLSTACK poses an overhead, if it is not needed. INVOCATION CONTEXTS can be used to make the CALLSTACK accessible from within a COMMAND LANGUAGE.

In the context of DYNAMIC METHODS the CALLSTACK plays another important role. Consider a situation in which a method is redefined while an invocation of this method is still running. For instance in the following example the running method is overwritten by a text read from a file. It has to be ensured that the final `close` statement of the original method invocation is reached in any case. This can be done by remembering the execution code (or compiled byte code) of an executed method from within the callframe. Reference counting for method bodies can be used to avoid the overhead of remembering a copy of the method body in each callframe.

```
C1 method x args {
  set FILE [open "method-x.def"]
  C1 method x args [read $FILE]
  close $FILE
}
```

Context An INTERPRETER is used for a COMMAND LANGUAGE.

Problem From within a COMMAND LANGUAGE script, information about the control flow is required at runtime, such as the currently executing object, class, and method or the calling object, class, and method.

Solution An INVOCATION CONTEXT is used to obtain the invocation information from inside of a running method. The INVOCATION CONTEXT contains at least information to identify the calling and called method, object, and class.

When a CALLSTACK is used, the INVOCATION CONTEXT is a view on the CALLSTACK and contains (at least) the information in the top-level callframe. In a COMMAND LANGUAGE, a COMMAND can be provided that allows scripts to obtain the INVOCATION CONTEXT from the CALLSTACK.

Figure 8 shows an INTERPRETER that puts each invocation onto a callstack. Thus from within a method `aMethod` it is possible to obtain the current INVOCATION CONTEXT.

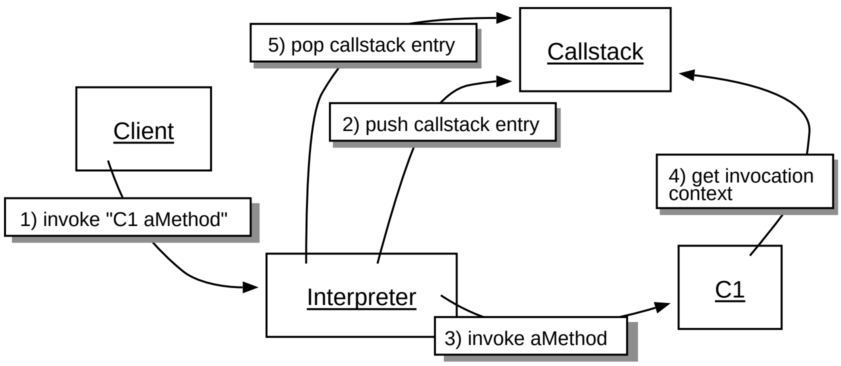

Figure 8. Invocation context obtained from a callstack

Discussion INVOCATION CONTEXTS are especially useful together with an INTERPRETER or other indirection techniques that require a CALLSTACK (or another way to maintain per call information). If the CALLSTACK information is not available, the system needs to be restructured to provide an additional indirection. This costs memory and performance.

3.7 Split Object

Context A COMMAND LANGUAGE is used within another language (the "host language").

Problem Typically the COMMAND LANGUAGE needs to access host language objects. Here "access" means performing lookups, invocations, creations, and destructions of objects and methods. These tasks can be handled by a wrapper object. Yet pure wrapping poses some problems in more complex language integration situations. A wrapper provides only a "shallow" interface into a system, and it does not reflect further semantics of the two languages. Examples of such semantics are class hierarchies or delegation relationships. Further, a wrapper does not allow one to introspect the system's structure. The logical object identity between wrapper and its wrappees is not explicit. Complex wrappers that are implemented by hand are hard to maintain.

Solution A SPLIT OBJECT is an object that physically exists as an instance in the COMMAND LANGUAGE and the host language, but logically it is treated like one, single instance. Both halfs can delegate invocations to the other half. One half is called the wrapper half, and it provides an automatic forwarding mechanism to send invocations to the wrappee half. The wrapper mimics the user-defined class hierarchy of the wrappee, variables are automatically traced and shared, and methods can be wrapped. Depending on the language features of the two languages, these functionalities can either be implemented by extending the language's dispatch process, using reflection, or using generative programming techniques.

Figure 9 shows a host language client that needs to invoke a host language object `Object1`. This object is a SPLIT OBJECT: instead of invoking this object directly, the counterpart in the COMMAND LANGUAGE is invoked first, which forwards the invocation back into the host language. This way the COMMAND LANGUAGE can intercept the invocation.

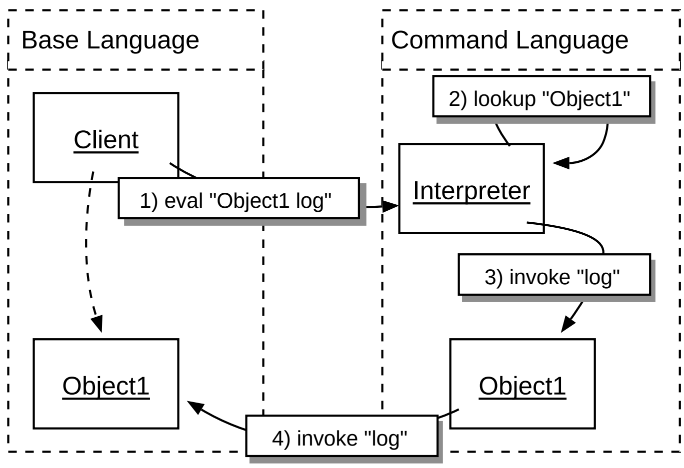

Figure 9. Invoking a split object through the command language

Discussion SPLIT OBJECTS can be used to deeply integrate two object systems. Concepts realized in one object system can be used from within the other object system. For instance, this way a DYNAMIC METHOD abstraction, implemented in the COMMAND LANGUAGE, can be used for host language objects.

SPLIT OBJECTS pose a memory and performance overhead and thus should only be used for host language objects that need to be accessed from the COMMAND LANGUAGE.

From the COMMAND LANGUAGE we can automatically forward invocations into the host language. In turn, ordinary host language invocations bypass the SPLIT OBJECT in the COMMAND LANGUAGE. Thus, from within the host language, we have to use the INTERPRETER'S `eval` method to access a SPLIT OBJECT. To avoid this additional invocation style, a HOOK INJECTOR can be used to replace host language invocations with indirections to SPLIT OBJECTS.

3.8 Hook Injector

Context A program text should be manipulated. For instance, invocations should be indirected into a COMMAND LANGUAGE.

Problem A sub-system's behavior should be modified, but neither the sub-system's nor its clients' code should be permanently changed. Consider you want to avoid invocations of the following style to deal with SPLIT OBJECTS:

```
interp.eval("MyObject create a");
interp.eval("a write Hello");
```

Instead all objects of the type `MyObject` should be made SPLIT OBJECTS, and all invocations should be sent through the COMMAND LANGUAGE first.

Solution Use a parser for the host language and let a HOOK INJECTOR inject the indirection hooks directly into the parse tree (or into the byte-code). Either write a custom compiler to directly create machine code or byte code, or, as a simpler alternative, produce a new program in the host language with the injected indirection hooks. Then let this program be compiled or interpreted, instead of the original program. Semantically the new code is equivalent to the original code, with the exception of the injected hooks for extracting or modifying the relevant invocations.

Figure 10 shows a HOOK INJECTOR that injects hooks by parsing a document, modifying the representation in memory (here a parse tree), and writing the modified source document back. This document is then interpreted or compiled, instead of the original source document.

Discussion In compiled languages, a HOOK INJECTOR only performs static modifications. Thus it is only possible to dynamically change those classes which are statically instrumented before. Instrumenting a class, for instance for

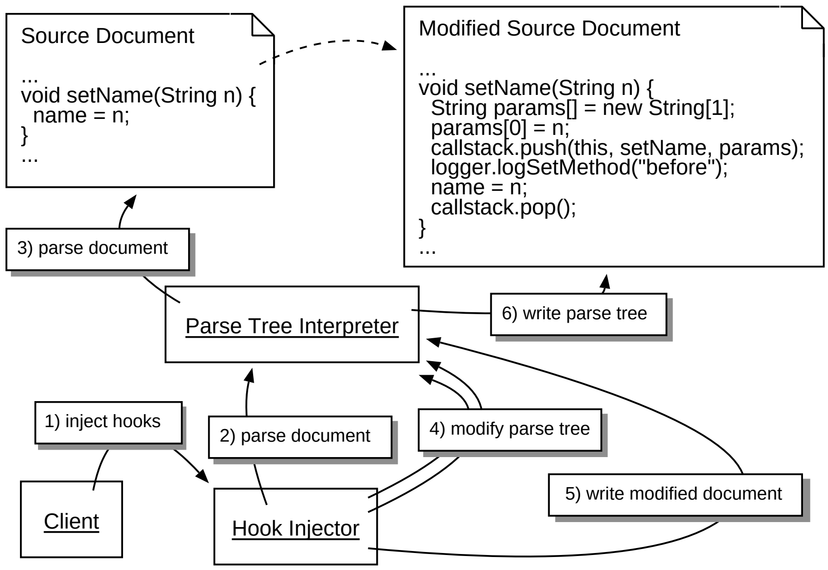

Figure 10. A hook injector manipulating the parse tree

introducing DYNAMIC METHOD, poses a performance and memory overhead, even if the DYNAMIC METHOD abstraction is not used later on.

4 Case Study: Design and Use of a Runtime Method Transformation Framework

In this section we present a case study of a method transformation framework implemented in Frag [30], an object-oriented extension of the programming language Tcl [10]. Frag is a full-fledged object-oriented programming language. However, it is not primarily designed for building complete systems, but it is rather intended as a composition and configuration language for other languages, namely C, C++, and Java. That is, Frag is typically embedded in these languages and it is used for configuring applications, for composing component architectures, or for providing a little, domain-specific language. To support such tasks, Frag offers a reflective and very flexible object system, and it provides means to be easily integrated with those other languages (examples are provided below). Frag specifically aims at Java because the Frag implementation is completely implemented in Tcl and runs in a Java Virtual Machine on top of Jacl [37] (of course, it also works with the standard Tcl implementation implemented in C).

Frag implements a DYNAMIC METHOD abstraction together with INTROSPECTION OPTIONS using the Tcl or Jacl INTERPRETER. This implementation is explained in Section 4.1. DYNAMIC METHODS can be used from other languages by using SPLIT OBJECTS. Using a HOOK INJECTOR we can instrument the other languages code to automatically invoke a SPLIT OBJECT half in Frag. We have implemented a HOOK INJECTOR for Java using AspectJ (see [38] for details) and HOOK INJECTORS for C++ using SWIG [39].

On top of the DYNAMIC METHOD abstraction, Frag provides a method transformation framework following the concepts from Section 2 (explained in Section 4.2).

In this section we provide two examples. Firstly, we provide a simple trace example. Secondly, we present a case study of using the transformation framework for interactive game scripting.

4.1 Frag's Dynamic Methods

Frag provides a highly flexible object system in which each element and relationship is completely dynamic and introspective. Objects can be interpreted differently in different contexts. That is why we call the Frag object system "contextual." An object might be *interpreted* as "more" than a pure object whenever the context makes it relevant. For instance, an object can play the role of a class or a superclass. The class concept of the language is not fixed but can be tailored to the particularities of a host language. This way Frag can easily be integrated with other languages such as C, C++, and Java (please refer to [30] for more details). In the remainder of this section we concentrate on DYNAMIC METHODS in Frag.

Frag's DYNAMIC METHODS are defined in a method table. Each object can have methods, but these are only applied when the object acts as a class for other instances. Consider we create a class `MyClass`. This class contains a simple method `writeMsg` which is defined by invoking the `method` operation and providing it with the method name, the parameter list, and the body of the method:

```
MyClass method writeMsg {msg} {
  puts $msg
}
```

All parameters passed to Frag methods are strings. Thus the parameter `msg` in the example method above has no type definitions in the signature, just the parameter name. SPLIT OBJECTS, wrapping statically typed host language objects, need to care for type conversions. Note that for performance reasons Tcl performs type conversions internally (e.g. an integer is internally stored and handled as an integer to avoid continuous back and forth conversion). But these internal data types and conversions are not visible to the language user. This simple, generic type concept for primitive data types is an important feature to let users who are not expert programmers understand the language concepts without sophisticated knowledge of a programming language's concepts.

The language element `self` can be used to refer to the current object. In Frag, methods can be redefined at arbitrary times by simply replacing the method definition. An invocation like the following in the same program as the invocation above, replaces the above method definition with the one below:

```
MyClass method writeMsg {msg} {
  puts "[self]: $msg"
}
```

We can also delete a method or rename a method using `rename`. Renaming a method to an empty string causes the method to be deleted. For instance, we can delete `writeMsg` as follows:

```
MyClass rename writeMsg {}
```

Frag provides a primitive `next` that implements *mixin methods*. That is, when `next` is called from within a method, all superclasses of the method's class are searched for the same-named method, and if it is found, it is invoked. Thus it is "mixed" into the current method execution. Dynamically classes can be added to and removed from the class hierarchy at arbitrary places. These classes are *mixin classes* containing mixin methods for extending the given class hierarchy. Mixin classes are a dynamic message interception techniques, and thus they are an alternative to runtime method transformations. We compare these two approaches in Sections 5 and 6.

Because Frag is designed for runtime composition, an important goal is to be able to find out the current composition of the objects (and classes) at any time. Therefore Frag is designed as a fully reflective language, offering INTROSPECTION OPTIONS for each language element it introduces. Introspection is realized by the method `info` of the class `Object`. `info` accepts a number of options. The following INTROSPECTION OPTIONS are relevant for DYNAMIC METHODS: `args` returns the parameter list of a method, `body` returns the body script of a method, and `methods` returns the list of methods defined for a class (for other INTROSPECTION OPTIONS see [30]).

For instance, the implementation of `copyMethod` of the `MethodTransformer` (introduced in the next section) uses the DYNAMIC METHOD abstraction to create a new method. Here, the INTROSPECTION OPTIONS options `args` and `body` are used to retrieve the arguments and body of the source method `srcMethod` on the source class `srcCl`:

```
MethodTransformer method \
  copyMethod {srccl srcmName targetcl targetmName} {
    $targetcl method $targetmName \
        [$srccl info args $srcmName] \
        [$srccl info body $srcmName]
```

```
}
```

Tcl (and thus Frag) is implemented as a COMMAND LANGUAGE. Every language element is a COMMAND. For instance, when a method is copied using `copyMethod` the DYNAMIC METHOD abstraction, accessed with `method`, creates a new METHOD COMMAND with the name `destMethod` for `destCl`. This COMMAND is bound to the implementation of the Frag METHOD COMMAND resolver and stores the arguments and body within the class. The Tcl INTERPRETER is used for interpretation of the METHOD COMMANDS, as well as for script evaluation. Tcl compiles methods internally using an on-the-fly byte-code compiler.

In Frag all invocations are pushed onto a CALLSTACK. This CALLSTACK is fully accessible from within the language. Using the Frag object `callstack` we can query the current INVOCATION CONTEXT (and all other INVOCATION CONTEXTS on the CALLSTACK). `self` is actually a short-cut for `callstack self`, which returns the top-level object on the callstack. `callstack method` returns the currently executing method, and `callstack class` returns the currently executing class. The options `callingObject`, `callingMethod`, and `callingClass` return the same information at the caller level.

For instance, the following method prints out the name of the object and method that have invoked it:

```
X method callerPrinter {} {
  puts "Invoked by [callstack callingObject],\
    [callstack callingMethod]"
}
```

4.2 Frag's Runtime Method Transformation Framework

In this section we explain Frag's runtime method transformation framework, following the concepts explained in Section 2. The main part of the runtime transformation framework is implemented in a `MethodTransformer` class. It contains methods for the runtime method transformation primitives as well as some convenience methods. As shown in Figure 11, the method transformer associates hash tables for intrinsic code, intrinsic arguments, before-code, after-code, and extrinsic arguments. The hash tables contain either script lists or argument lists. The argument lists contain the arguments in the order in which they should be applied, whereas the script list contain the scripts in the order in which they should be composed. We do not maintain return types, as there is only one generic return type in Tcl (i.e. strings).

The method transformer class stores the DYNAMIC METHOD definitions (arguments and code snippets) in the hash tables. The combinations of method name and class name are used as keys for the hash tables. Thus for any com-

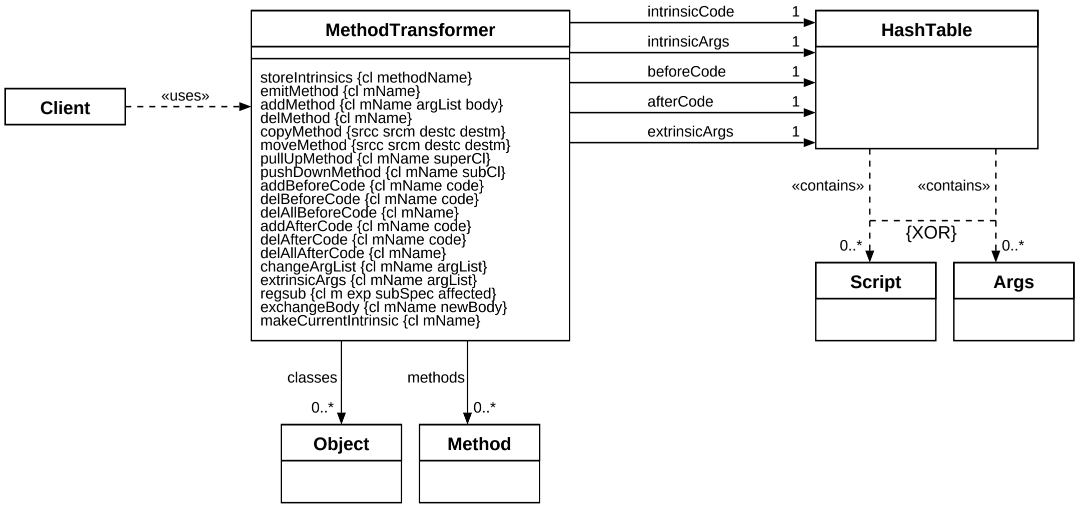

Figure 11. Method Transformer Design

bination of method and class we can store the intrinsics, a list of before-code fragments, a list of after-code fragments, and extrinsic arguments.

Before a non-intrusive method transformation is performed the first time, it is ensured that the intrinsics are stored in the two intrinsic hash tables. Thus the intrinsic method definition is saved.

A convenience method `emitMethod` is used to construct the current method definition from before-code, after-code, and intrinsics. It is invoked after each change to method extrinsics or intrinsics.

A method `makeCurrentIntrinsic` is supported to make the current method definition the intrinsics, so that any change performed so far is accepted. Before this method is invoked, it is always possible to go back to the original definition by making the intrinsics the current method definition.

The rest of the method transformer code are the individual method transformation primitives, as they have been explained already in Section 2. Thus we do not repeat these here; the individual method names can be seen in Figure 11.

Besides support for incremental and experimental software evolution, there are four main uses for the method transformer framework in our work:

- *Transformation and Evolution:* The method transformation framework can be loaded into any Frag system and be used as a high-level program transformation technique. That is, (existing) methods can be transformed, refactorings can be applied, and the system can be incrementally evolved.
- *Object-Oriented Adaptation:* As pointed out in Section 5 many object-oriented adaptation techniques rely on static method transformations. The

method transformer framework can be used to implement similar dynamic techniques. The method transformations are then used as a language extension for runtime object-oriented adaptation. It is possible to perform structure adaptations (similar e.g. to AspectJ introductions) as well as behavior adaptations (similar e.g. to AspectJ pointcuts and advices).

- *Optimization:* A special use of the method transformer is optimization of message interceptor code. Frag classes can be used as mixin classes for other classes. This implies a dynamic dispatch of methods to the mixin class. At performance bottlenecks it might be better to use native, byte-code compiled methods without any further dispatch. Still in these situations it makes sense to use message interceptors as a design abstraction. Here, we can apply the method transformer: we still design and implement the application using the message interceptors, but where a performance bottleneck arises, we use the method transformer to inline the message interceptor code in the calling methods dynamically (see Section 6 for a performance comparison).
- *Tools:* We also use the method transformer in a component composition and reengineering tool for experimentation purposes.

4.3 Trace Example

As an example, let us consider a typical trace example (as often used as an example in the field of aspect orientation). Consider there is a set of classes `Circle`, `Point`, and `Square`, derived from a generic `Figure` class and we want to trace any setter method (methods starting with `set`) of these classes. But we do not have any knowledge which setter methods are defined on which subclasses, what might even change when more subclasses are defined.

As a solution, we derive a class `SetterTrace` from the `MethodTransformer` class. This class has a method `traceAllSetters` for tracing the setter methods on all subclasses with a specified piece of before-code. First we have to find all relevant classes, what is done by a simple recursive method using INTROSPECTION OPTIONS. Then we iterate over all subclasses and over all methods of these subclasses with two `foreach` loops. For every method name that begins with `set`, we add the specified before-code:

```
Object create SetterTrace \
  -superclasses MethodTransformer \
  -makeSelfClass
SetterTrace method traceAllSetters {cl code} {
  set classes [concat $cl [$cl getAllSubclasses]]
  foreach c $classes {
    foreach method [$c info methods] {
      if {[string match set* $method]} {
        self addBeforeCode $c $method $code
```

```
      }
    }
  }
}
```

Next, we can apply the setter trace to all classes that are derived from `Figure`:

```
SetterTrace traceAllSetters Circle {
  puts "CALL Class=  [callstack class]"
  puts "     Method= [callstack method]"
}
```

Now every setter method on every figure class prints the class and method name before it is executed.

4.4 End-User Programming Example: Interactive Game Scripting

Consider the example of developing game scripts for interactive games that should run on the digital television set-top box for the multimedia home platform (MHP) [40]. Specifying in-game scenes and character behavior is a complex task, and thus a programming language or domain-specific language is useful here. MHP settop boxes run Java programs. Yet programming in-game scenes in Java is problematic, because game level and scene designers usually are not professional programmers. Thus, what is need, is a simple configuration language that can easily be connected to those elements of the Java program which are relevant for game scripting. In such cases, we propose to use SPLIT OBJECTS and a dynamic COMMAND LANGUAGE. For instance, Frag is designed for configuring Java using scripts.

Consider, for instance, a Java class `Wizard` provides all basic actions for a wizard character, such as character painting, move sequences, spell cast movements, etc. Now consider further the wizard is capable of some 100 spells, each having different effects on the wizard and the spell's target. Also each spell causes different visual effects. Configuring these spells is a typical game scripting task. For instance, a spell script might look as follows:

```
JavaClass create Wizard -superclasses Character
...
Wizard method castBurnSpell {target} {
  self spellCastMovement 3
  set success [self castSpell fireball]
  self substractMana 15
  $target burn [expr 2 * $success]
  $target hit [expr 3 * $success]
}
```

A few scripts might be provided by the game engine programmer as an example. The rest of the scripts should be developed by game level and scene designers. Obviously, many parameters in this script (and all other spell scripts) need extensive game playing and testing by trial and error. If it would be necessary to re-compile and re-start the game application to change parameters or behavior of the scripts, there would be a considerable overhead in terms of development times. Instead it makes sense to dynamically manipulate and exchange the scripts during game play testing.

The SPLIT OBJECT solution allows us to untangle the aspect "in-game configuration" from the game code. Other AOP solutions would also work in this context, but as a disadvantage many current AOP languages require recompilation. The SPLIT OBJECT solution has the disadvantages that the embedded INTERPRETER is slower than a compiled solution. For character scripts and in-game scenes this loss of speed can be tolerated. The game engine, however, should be developed in Java.

Using the runtime method transformation framework, the game level and scene designers can simply use the existing examples and change it by trial and error. For instance, one can copy the given example to a new method and then go into this method's definition and manipulate it:

```
MethodTransformer copyMethod Wizard castBurnSpell \
  Wizard castFireWallSpell
```

This way the concept of exemplification can be supported by a runtime method transformation framework (what lowers the learning curve).

Consider there is simple callback test routine defined to run one test movement of a character:

```
Core method testVisual {} {;}
```

This method can be manipulated to run different tests using the `exchangeBody` and `regsub` operations. For instance, we can test the new spell on a `Paladin` like this:

```
MethodTransformer exchangeBody Core testVisual {
  Paladin create p1
  Wizard create w1
  w1 castFireWallSpell p1
}
```

Another example of method transformation during game scripting is adding before-code or after-code for observation tasks. This way it is very easy to design the interaction of characters. For instance, group behavior can be implemented by adding notifications of the character objects as before-code or after-code of the group object's methods.

Note that it is not intended to let the game level and scene designers send the invocations to the `MethodTransformer` by hand (as in the examples above). Instead a simple programming tool can provide the `MethodTransformer's` primitives within a GUI (see [41] for examples of how such a tool can look like). The game level and scene designers only need to provide the parameters and method bodies. The tool can also check that game level and scene designers do not manipulate methods or classes that they should not use.

The `MethodTransformer` framework provides the powerful adaptation technique implemented by DYNAMIC METHODS in a simplified and safe manner to the game level and scene designers (here using a simple GUI). The core technical solution, comprising the SPLIT OBJECT solution and Frag's DYNAMIC METHODS, works under the hood of the `MethodTransformer` framework. Thus, the `MethodTransformer` framework is intentionally very simple, whereas the pattern concepts used in the implementation are in comparison rather complex. The use of the two-language concept has the advantage that script developers can solely develop in the scripting language which is designed for rapid application development and offers a low learning curve, whereas the core application logic can be developed in the system language Java.

5 Related Work

In this section, we discuss some related work to put this paper into context. First, we discuss related approaches based on static transformations briefly. Next, we discuss other approaches for system evolution at runtime.

5.1 Static Transformation Approaches

Static transformation refers to approaches transforming a system before runtime, for instance, at compile time or load time. There are mainly two relations of runtime method transformations to static transformation approaches:

- Runtime method transformations can implement most of the tasks performed by static transformation approaches; thus they are an alternative for static approaches in situations where runtime variation is required.
- If a system is written in a language without support for DYNAMIC METHODS, static transformation approaches can be used to prepare the system for runtime method transformation (as explained in the HOOK INJECTOR pattern).

Static transformation approaches with these characteristics are applied for many different tasks; some example approaches are:

- Different aspect-oriented approaches [15,16,17,18] use static method transformations of source code or byte code for their internal implementation (see [28] for a discussion of implementation details of AOP frameworks).

- A number of static transformation techniques are developed in the context of software maintenance and reengineering scenarios [21].
- Generative programming [42] refers to systems that generate customized components utilizing modifications of given code fragments and component assembly apart from given patterns.
- Partial evaluation [43] creates a specialized version of a general program and can be implemented using program transformations.
- Wrapping is often used to migrate an existing system to a new technology or programming language [44]. Static wrappers are often generated using wrapper generators (see e.g. [39]).

Besides these static transformation approaches, there are many other approaches for modifying a system at runtime (discussed in the next sections).

5.2 Wrappers, Mixins, Message Interceptors, and Composition Filters

Moon proposed flavors as small units of composition that can be mixed into a given class hierarchy at arbitrary places [45]. Subsequently, mixins were proposed for instance in CLOS [3]. Mixins are classes whose superclass is not specified at mixin implementation time, but is left to be specified at mixin use time. An automatic method combination mechanism (such as `call-next-method` in CLOS) is used within the mixin. In CLOS mixins are rather a coding convention; mixin-based inheritance [46] proposes them as formal construct.

An alternative for static wrappers, similar to the mixin concept, are dynamic wrapping approaches, such as method wrappers [47]. In this concept, invocations of a wrapped method are indirected to the method wrapper first. The method wrapper (optionally) forwards the invocation to the original method implementation, and it can execute code before and/or after the invocation of the original method. Thus no transformation of the original method implementation is necessary.

An extension to method wrappers and simple mixin concepts are message interceptors. Message interceptors are sophisticated wrappers that introduce new behavior to be executed before, after, and/or around an existing method or component at runtime. The most important conceptual extensions, introduced by interceptors, are that they can be ordered in chains (or other structures) and provide some (semi-)automatic forwarding mechanism. Support for message interceptors can be provided in different environments. For example, XOTcl [25] is an object-oriented scripting language that supports message interception by special language constructs. Also, interceptors for distributed invocations are provided in various middleware systems, such as TAO [27] or Orbix [26]. The INTERCEPTOR pattern [48] describes this form of message interception.

We have already explained that method transformations are an alternative to the forwarding approach chosen in method wrappers, mixins, and message interceptors. Runtime method transformations can be used to optimize these techniques. We have already described that Frag supports mixin methods and mixin classes as a message interception technique (we perform a performance comparison with runtime method transformations in Section 6.2). Frag supports both, mixin classes and runtime method transformations, because they are complementary adaptation concepts. In design situations that require runtime adaptation and separation of concerns at the same time, mixins or message interceptors tend to be the more suitable design abstraction, because they are separated units. In other situations, such as those of experimental or incremental program evolution, runtime method transformations are usually better suited, because they apply changes directly to the design unit (here: the method) to be incrementally evolved.

Composition filters [49] are a declarative model that explains the basic adaptation concept behind mixin classes and interceptors. The model realizes object composition using input and output filters for objects and classes. When an object receives a message, first an input filter chain is traversed, then the object implementation is invoked, and finally an output filter chain is traversed. Lately the model was extended to support aspect-oriented concerns as composition filters. Composition filters can be used for incremental and experimental transformation of methods by introducing filters that override or adapt a method invocation. Composition filters are an alternative for the runtime method transformation primitives that modify before or after code. They can easily express extrinsic around behavior which is difficult to achieve with method transformations, as discussed in Section 2.2. But intrinsic changes of a method, such as the method rewriting primitives, are not supported by composition filters.

5.3 Approaches for Dynamic Aspect Weaving

Lämmel proposes method call interception (MCI) [22] as a model for superimposing extra functionality onto method calls at runtime. A semantics-directed implementation of MCI is proposed in [50]. The aspects allow for runtime adaptation. A central registry for aspects indirects invocations to classes that 'superimpose' behavior onto other classes. Runtime method transformation primitives that do not change the method intrinsics can directly be implemented with MCI. The implementation of the prototype uses static method transformation following the HOOK INJECTOR pattern: hooks are injected into each method of the respective classes.

There are a number of other approaches that use the HOOK INJECTOR approach for preparing a system for dynamic aspects in Java. For instance, JBoss AOP [51] modifies the class loader to insert hooks into the Java byte code at load time, and the aspect composition can be changed at runtime. Prose [24] performs a similar modification by modifying the native code compilation performed by the Jikes virtual machine at runtime.

AspectS [23] provides a runtime aspect weaver for Smalltalk. It uses the method wrapper concept, explained above, and adds method wrappers to a program using meta-programming techniques. In particular, method wrappers replace an entry in a class method dictionary, add behavior to the method invocation, and eventually invoke the wrapped method itself.

5.4 Approaches Utilizing Dynamic Methods

There are many approaches utilizing DYNAMIC METHOD abstractions. We have already explained that a number of scripting languages, including Tcl [10], Python [11], Perl [12], and Ruby [13], offer DYNAMIC METHODS natively. In contrast to our work, these languages do not provide further conceptual support for runtime method evolution. The patterns described in this paper can be implemented in all these languages in a similar way as in our case study.

Heinlein implements DYNAMIC METHODS in pure Java using a pre-compiler based language extension and a Java-like syntax for DYNAMIC METHODS [31]. This approach is an alternative to the two-language approach proposed in our case study. We have used the two-language approach in order to benefit from the language diversity. Users only need to learn a sub-set of the simple scripting language leading to a low learning curve. Moreover, the Frag implementation can be reused for other languages than Java as well. However, the more Java-like syntax of Heinlein's approach might be more appealing to experienced Java developers. Note that Heinlein's approach is also covered by our pattern language (it just omits the pattern SPLIT OBJECT).

The Refactoring Browser [19] is a Smalltalk browser with support for refactorings implemented as dynamic method changes. That is, a developer can automatically perform behavior preserving transformations, as discussed in [20]. Also there are some rewrite tools for source-to-source transformations that can be accessed from the tool.

6 Experiences and Evaluation

6.1 Consequences of Applying Runtime Method Transformations

Our approach offers a unique combination of the following properties (in the related works discussed in Section 5 one or more of these properties are missing):

- *Runtime evolution:* Typical application areas of our approach require experimentation and incremental evolution at runtime. Using static transformation or composition techniques is rather cumbersome here.
- *Simplicity:* Most object-oriented adaptation techniques have rather complex models that are hard to understand for non-expert programmers. Method transformations, in contrast, offer a simple, yet powerful concept that can quickly be fully understood by someone who is not an expert programmer. Our approach only requires the user to understand the local context in which (s)he works plus the limited number of method transformation primitives. These can be applied safely (meaning that any change can be discarded, if it was not successful). It is easy to provide tool support and integrate rapid application development environments, such as scripting languages, with our concepts.
- *Language diversity:* Most implementations of object-oriented adaptation techniques focus on a single programming language only, and thus cannot be used with another language. Using the SPLIT OBJECT approach, we can combine our method transformation framework with any language that can work with Frag.
- *Learning by example:* The concept of exemplification is a typical way to allow novices to quickly learn how to modify a software system without fully understanding it. One can use similar existing methods (see Section 4.4 for an example) and modify it slightly. Learning by trial-and-error is always limited in its capabilities, but it can motivate users to learn the full languages or interfaces of the system. Typical object-oriented adaptation techniques do not support exemplifications.
- *Experimentation:* In typical object-oriented adaptation techniques, it is hard to try out ideas or "play" with a system, say, because recompilations and restarts of the system are necessary for each change. In contrast, our concepts are designed for experimentation with a system while it runs.
- *Memory and performance:* Almost all object-oriented adaptation techniques have a considerable overhead in terms of performance and memory consumption compared to native object-oriented methods. As discussed below, runtime method transformations have only a low overhead and can be used to optimize mixin class performance.

There are also a set of characteristic drawbacks that should be considered before applying the concepts proposed in this paper.

All runtime method transformations that are applied using INTROSPECTION OPTIONS are only applied for the current state of the system, and not for future changes. For instance, when all methods of a class C are transformed, and then a new method is added to C, this method is not automatically transformed as well. In contrast, for instance, a mixin class can observe the `addMethod` method of the method transformer, and trigger transformations if necessary.

The implementation of a framework using the pattern language is a straightforward task, if an existing INTERPRETER is reused, as in our examples Tcl [10] and Jacl [37]. Implementing a full-fledged general-purpose COMMAND LANGUAGE from scratch can be a huge effort; however, it is also possible to implement a little, domain-specific COMMAND LANGUAGE (what is much less an effort). Note that it is not necessary to use the two-language approach to work with our concepts (we have already discussed that Java dynamic methods [31] could be used instead). In general, SPLIT OBJECTS have a performance and memory penalty, because objects are implemented in two languages and some invocations need to be dispatched twice.

As explained in the related work section, runtime method transformations are good for solving experimental problems, but where class-like abstractions are helping to solve the problem, aspects or mixin classes might be the better – more understandable – abstraction, as they group related adaptations in one computational entity. A solution to this problem is to support both abstractions, as in Frag.

6.2 Performance Evaluation

Regarding performance we have predicted that our runtime method transformation framework can be used for optimization of dynamic interceptor techniques and that it does not perform significantly different to delegation. To verify this claim, we have performed a performance comparison using a simplistic example.

The example instantiates a simple circle class with variables for radius, and x and y coordinates. We have timed two method invocations, invoking a method `perimeter` and a method `area`, performing the respective simple computations. A trace object is used to print trace messages, and the trace is invoked before and after the method invocation. We have stripped all outputs, so that only the invocations are measured.

We have measured the time for invoking the two circle methods without trac-

Test	No Instrumentation	Hand-Built Delegation	Method Transformation	Mixin Class
1 Trace: Before/After Tracing	53 ms	66 ms	66 ms	129 ms
1 Trace: Registration/Transformation	–	–	253 ms	202 ms
3 Traces: Before/After Tracing	53 ms	86 ms	87 ms	265 ms
3 Traces: Registration/Transformation	–	–	646 ms	260 ms

Table 1
Performance comparison: Times in milli-seconds

ing. Next, we have measured the time to transform a method and the invocation times of the traced methods. Finally, we have used a Frag mixin class for the same trace functionality. To be applied for tracing, this message interceptor needs to be registered as a class first.

The results are summarized in Table 1. All results are measured in milliseconds. All measurements were performed on an Intel P4, 2.53 GHz, 1 GB RAM running Red Hat Linux 8.1. Frag version 0.26 together with Tcl 8.4 was used.

We can see that before/after tracing with runtime method transformation and delegation are almost identical, as expected. Both only add a slight overhead to the version without instrumentation. In contrast to these solutions the mixin class requires an additional method dispatch and the `next` primitive has to be resolved. This results in a substantial overhead of 95% for one trace call. The mixin class registration is 20% faster than the method transformation. For three traces on the same methods, the method transformation is only slightly slower than for one trace. Three consecutive method transformations consume about three times as much time as one trace. The mixin class registration does not add much overhead for three traces, but three mixins require considerably more time to be invoked than one mixin. Thus, method transformations can well be applied to optimize the performance of runtime message interceptors.

Note that, even though our results are encouraging for the use of runtime method transformations in fields that require incremental, experimental, or highly dynamic software evolution or adaptation, there are also other areas, where our results indicate that runtime method transformations should not be applied. For instance, where no dynamic transformation is required, static techniques still perform better because they have no runtime costs for transformation.

7 Conclusion

In this paper we have proposed an approach for runtime method transformations that can be applied automatically. The approach covers a runtime method transformation concept that primarily simplifies and safely applies dynamic methods, a pattern language for implementing such runtime method transformations, and a case study of a runtime method transformation framework. Our approach is unique in a number of ways. In contrast to many other object-oriented adaptation techniques (e.g. from the AOP field), adaptations are performed at runtime. In contrast to meta-level or reflective approaches, the approach is very simple. To our knowledge there is no comparable approach that focuses on multiple languages. The framework is part of a mature language implementation, and the concepts are applied in a number of projects. The primary application areas for our approach are different fields of experimental and incremental program evolution. But as pointed out in the paper the approach can also be applied for other areas, such as optimizing the performance of message interceptors and implementing dynamic composition of aspects. Note that the approach should only be applied if dynamic program evolution is required, because building a DYNAMIC METHOD infrastructure is a substantial work, if no existing implementation can be reused. Especially if a HOOK INJECTOR and SPLIT OBJECTS are used for introducing DYNAMIC METHODS there is an overhead in terms of memory and performance consumption compared to native methods.

Acknowledgments

The author likes to thank Ralf Lämmel for his helpful comments on this paper.

References

[1] G. Kniesel, J. Noppen, T. Mens, J. Buckley, The first workshop on unanticipated software evolution (USE 2002), in: ECOOP 2002 Workshop Reader, Springer Verlag, LNCS 2548, 2002.

[2] C. Letondal, U. Zdun, Anticipating scientific software evolution as a combined technological and design approach, in: Proceedings of Second International Workshop on Unanticipated Software Evolution (USE 2003), Warsaw, Poland, 2003.

[3] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon, Common Lisp Object System Specification, Sigplan Notices 23 (9).

[4] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison Wesley, Reading, MA, 1989.

[5] D. Ungar, R. B. Smith, Self: The power of simplicity, in: Proc. of OOPSLA '87, Orlando, 1987, pp. 227–242.

[6] B. Smith, Reflection and semantics in Lisp, in: Eleventh Annual ACM Symposium on Principles of Programming Languages (POPL), Salt Lake City, Utah, 1984, pp. 23–35.

[7] P. Maes, Concepts and experiments in computational reflection, ACM SIGPLAN Notices 22 (12) (1987) 147–155.

[8] G. L. Steele, Common Lisp: The Language, Digital Press, Bedford, MA, 1984.

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

[10] J. K. Ousterhout, Tcl: An embeddable command language, in: Proc. of the 1990 Winter USENIX Conference, 1990, pp. 133–146.

[11] G. van Rossum, Python reference manual, Tech. Rep. CS-R9525, CWI, Amsterdam, The Netherlands (May 1995).

[12] L. Wall, T. Christiansen, R. L. Schwartz, Programming Perl, 2nd Edition, O'Reilly & Associates, 1996.

[13] Y. Matsumoto, Ruby in a Nutshell, O'Reilly & Associates, 2001.

[14] J. K. Ousterhout, Scripting: Higher level programming for the 21st century, IEEE Computer 31 (1998) 23–31.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of ECOOP'97, LCNS 1241, Springer-Verlag, Finnland, 1997, pp. 220–242.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, Getting started with AspectJ, Communications of the ACM 44 (10) (2001) 59–65.

[17] P. Tarr, Hyper/J, http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm (2003).

[18] S. Chiba, Javassist, http://www.csg.is.titech.ac.jp/ chiba/javassist/ (2003).

[19] J. Brant, Refactoring browser, http://st-www.cs.uiuc.edu/users/brant/Refactory (2003).

[20] M. Fowler, Refactoring – Improving the Design of Existing Code, Addison Wesley, Reading, MA, 1999.

[21] E. Visser, A survey of strategies in program transformation systems, Electronic Notes in Theoretical Computer Science 57.

[22] R. Lämmel, A Semantical Approach to Method-Call Interception, in: Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), ACM Press, Twente, The Netherlands, 2002, pp. 41–55.

[23] R. Hirschfeld, Aspects – aspect-oriented programming with squeak, in: Objects, Components, Architectures, Services, and Applications for a Networked World, LNCS 2591, Springer-Verlag, 2003, pp. 216–232.

[24] A. Popovici, T. Gross, G. Alonso, Just In Time Aspects: Efficient Dynamic Weaving for Java, in: Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), ACM Press, Boston, USA, 2003, pp. 100–109.

[25] G. Neumann, U. Zdun, XOTcl, an object-oriented scripting language, in: Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference, Austin, Texas, USA, 2000, pp. 163–174.

[26] IONA Technologies Ltd., The orbix architecture (August 1993).

[27] N. Wang, K. Parameswaran, D. C. Schmidt, The design and performance of meta-programming mechanisms for object request broker middleware, in: Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), San Antonio, TX, USA, 2001.

[28] U. Zdun, A pattern language for the design of aspect languages and aspect composition frameworks, Draft; accepted for publication in IEE Proceedings Software, special issue on Unanticipated Software Evolution (2004).

[29] B. Nardi, A small Matter of Programming: Perspecives on End User Computing, MIT Press, 1993.

[30] U. Zdun, Frag, http://frag.sourceforge.net/ (2003).

[31] C. Heinlein, Dynamic class methods in Java, Tech. Rep. Ulmer Informatik-Berichte 2003-5, University of Ulm, Ulm, Germany (July 2003).

[32] B. B. Kristensen, K. Østerbye, Roles: Conceptual abstraction theory & practical language issues, Theory and Practice of Object Systems 2 (1996) 143–160.

[33] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer, Simple object access protocol (SOAP) 1.1, http://www.w3.org/TR/SOAP/ (2000).

[34] C. Alexander, The Timeless Way of Building, Oxford Univ. Press, 1979.

[35] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[36] U. Zdun, Patterns of tracing software structures and dependencies, in: Proceedings of EuroPlop 2003, Irsee, Germany, 2003.

[37] M. DeJong, S. Redman, Tcl Java Integration, http://www.tcl.tk/software/java/ (2003).

[38] U. Zdun, Using split objects for maintenance and reengineering tasks, in: 8th European Conference on Software Maintenance and Reengineering (CSMR'04), Lisbon, Portugal, 2004.

[39] Swig Project, Simplified wrapper and interface generator, http://www.swig.org/ (2003).

[40] ETSI, MHP specification 1.0.1, ETSI standard TS101-812 (October 2001).

[41] C. Letondal, Programmation et interaction, Ph.D. thesis, Université de Paris XI, Orsay (2001).

[42] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Techniques and Applications, Addison-Wesley, 1999.

[43] N. Jones, C. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993.

[44] H. M. Sneed, Encapsulation of legacy software: A technique for reusing legacy software components, Annals of Software Engineering 9.

[45] D. Moon, Object-oriented programming with flavors, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86), Vol. 21 of SIGPLAN Notices, Portland, 1986, pp. 1–8.

[46] G. Bracha, W. Cook, Mixin-based inheritance, in: Proc. of OOPSLA/ECOOP'90, Vol. 25 of SIGPLAN Notices, 1990, pp. 303–311.

[47] J. Brant, R. E. Johnson, D. Roberts, B. Foote, Evolution, architecture, and metamorphosis, in: Proc. of 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, 1998, pp. 396–417.

[48] D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Patterns for Concurrent and Distributed Objects, Pattern-Oriented Software Architecture, J. Wiley and Sons Ltd., 2000.

[49] L. Bergmans, M. Aksit, Composing crosscutting concerns using composition filters, Communications of the ACM 44 (10) (2001) 51–57.

[50] R. Lämmel, C. Stenzel, Semantics-Directed Implementation of Method-Call Interception, 46 pages; Accepted for publication in IEE Proceedings Software; Special Issue on Unanticipated Software Evolution (Nov. 2003).

[51] B. Burke, JBoss aspect oriented programming, http://www.jboss.org/developers/projects/jboss/aop.jsp (2003).