Tailorable Language for Behavioral Composition and Configuration of Software Components

Uwe Zdun

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

Abstract

Many software systems suffer from missing support for behavioral (runtime) composition and configuration of software components. The concern “behavioral composition and configuration” is not treated as a first-class entity, but instead it is hard-coded in different programming styles, leading to tangled composition and configuration code that is hard to understand and maintain. We propose to embed a dynamic language with a tailorable object and class concept into the host language in which the components are written, and use the tailorable language for behavioral composition and configuration tasks. Using this approach we can separate the concerns “behavioral composition and configuration” from the rest of the software system, leading to a more reusable, understandable, and maintainable composition and configuration of software components.

Key words: Software components, component composition, component configuration, tailorable language.

1 Introduction

Many software systems depend on a behavioral composition and configuration of their software components. The *composition* of software components defines how the components of a

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

software architecture are plugged together; i.e. which components can interact in which way with which other components. For instance, Java classes declare their `import` to indicate which other classes are required by the class. In the following example we import an EJB from another EJB using two Java `import` statements:

```
import stocks.StockQuotes;
import stocks.StockQuotesHome;
```

Note that this is only a declarative composition at compile time. The *behavior* of the composition – i.e. how the components/classes are plugged together at runtime – is handled by code that is part of the importing EJB:

```
Context ctx = new InitialContext();
StockQuotesHome = (StockQuotesHome)
  ctx.lookup("java:comp/env/ejb/stocks");
```

In addition to component composition, component configuration is required. The *configuration* of a software component defines the runtime properties of one concrete component instance. For example, we can define simple properties of a component instance, such as an human-readable name or a textual description of the component (such properties are called metadata or annotations). We can also configure more complex properties. For instance in the above example for each EJB instance we need to configure to which `StockQuotes` instance it is bound in which way. Today such configurations are often given in a declarative manner, for instance, in XML. In the EJB example the importing EJB's deployment descriptor references the `StockQuotes` instance:

```
<ejb-ref>
  <description>Reference to stock quotes EJB</description>
  <ejb-ref-name>ejb/stocks</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>stocks.StockQuotesHome</home>
  <remote>stocks.StockQuotes</remote>
  <ejb-link>StockQuotes</ejb-link>
</ejb-ref>
```

Even though different component approaches realize component composition and configuration in different ways, the general solutions are similar. (Note that many industrial systems use no explicit component composition and configuration concepts at all. Thus the concern "composition and configuration" is not separated in a deployment descriptor, but instead it is completely hard-coded in the system's code).

The main problem that this paper deals with is that *behavioral composition and configuration* is not supported and thus must be provided using hard-coded program fragments. That is, the runtime semantics of the composition or configuration relationship must always be foreseen by the developer. Any kind of unanticipated software evolution [1], related to the composition and configuration code, requires substantial amounts of changes to the software system. Unfortunately, unanticipated changes are frequently required. Just consider we want to introduce a conditional composition or configuration in the above example: We would have to hard-code the condition as part of the business logic.

In some systems, the situation is even more severe: behavior changes are required and cannot be foreseen until *runtime*. Runtime behavior composition and configuration, however, is not supported.

A consequence of missing support for behavioral composition and configuration support is that the principle separation of concerns is violated: The concerns "composition and configuration" are *tangled* in the business logic. Tangled code makes the business logic hard to understand and maintain, and the concern "behavioral composition and configuration" becomes hard to reuse [2].

In summary, the problem of missing support for behavioral composition and configuration has a significant impact on important quality attributes of a software architecture [3], including the maintainability, understandability, reusability, and flexibility. In this paper we propose a language-based approach to solve these problems. In particular, we propose to implement the pattern Object System Layer [4] on top of a dynamically interpreted language and to embed this language in the language in which the components are written. In the Object System Layer we implement a dynamic object system, reflection techniques, and support for flexible classification (using dynamic mixin-based inheritance, dynamic classes, dynamic aspects, or any equally powerful adaptation concept). The resulting language/architecture is called a *tailorable language* because it can be adapted to different composition and configuration tasks in different environments. It is used for the behavioral composition and configuration concerns in the system. We claim (and later provide evidence for these claims) that this approach has the following contributions:

- Our concept, presented in Section 3, supports behavioral runtime composition and configuration in an easy-to-use manner. Also integration with languages or component frameworks is easy to achieve. We demonstrate this using an example from a case study in Section 4, a prototype implementation in Section 5, and an example how the prototype is integrated in Java and C++ in Section 6.
- The concerns "composition and configuration" are separated from the business logic, and thus a better separation of concerns is achieved.
- Our approach re-uses a few well-known, but non-simple concepts, such as dynamic and homoiconic languages, dynamic object systems, mixin concepts, etc. These are not easy to build from scratch, but fortunately many elements are already implemented and can be reused. To demonstrate this we have implemented a prototype language, Frag, that implements these concepts (see Section 5). We discuss the details of this implementation because for us it was

not obvious which elements are required for a well-working solution implementing the concepts from Section 3. The trade-offs of our prototype are briefly evaluated in Section 7. We also hope this discussion helps readers to realize similar architectures in other languages or environments.

- The concepts on which our approach is based have been defined in various areas, such as scripting, aspect-orientation, language integration, component-orientation, reengineering, and others (more details are discussed and contrasted to our approach in Section 8). Surprisingly the concepts have not been combined to a language-based approach for behavioral runtime composition and configuration. We claim that applying the concepts *together* is important for improving the way we compose and configure components with behavior.

2 Background and motivating examples

To set the scene for this paper, we want to briefly introduce the notion of "components" used throughout this paper. Then we give some motivating examples of systems in which the above introduced problem of missing behavioral configuration and composition support is of importance.

2.1 The component concept

In this paper we use the definition of the term *software component* by Szyperski [5]:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

This definition is quite broad and includes as diverse entities as server components (such as EJB, CCM, and COM+), Java Beans, component frameworks in scripting languages (such as Tcl, Python, Perl, and Visual Basic), Active X controls, C libraries with distinct APIs, etc. Note that the industrial understanding of the term component is often still quite different to an "ideal" academical understanding: Industrial components are often of a large scale, with no enforced boundaries, and only loosely defined interfaces [6]. Thus the notion of a component is not really well defined for practical purposes because the term is used to denote many different things (see also [7]). For the purposes of this paper, we do not want to exclude any of these meanings of a software component.

Though all named kinds of component offer some means for component composition or configuration, none of the above named component concepts focuses on this aspect specifically or does solve the problems motivated in Section 1 completely.

2.2 Motivating examples

Now that we have defined our understanding of the term software component, let us give a few motivating examples from real world systems in which behavioral composition and configuration of components is needed, but not well-supported.

In many projects, multiple platforms and environments have to be supported, including various operating systems, different hardware platforms (e.g. PCs, embedded devices, settop boxes), and software platforms (e.g. standalone, application server). These come with different characteristics (e.g. performance, threading, memory). Components might have to be configured flexibly to different platforms, and re-compilation is not always possible. It is significantly easier to specify such configurations in a behavioral fashion than to use declarative configuration options because for each change, new configuration options have to be defined and deployed to clients.

Consider, for instance, the TPMHP project [8,9], developing a product line architecture for the Multimedia Home Platform (MHP) [10]. Components are broadcasted to platforms, such as digital settop-boxes, PCs, and TV sets. That is, the component has to be composed and configured with the components that are already present on the platform. This has to happen at runtime: when the component arrives at the platform and got started. For instance, different compositions or configurations are required for different hardware capabilities on the settop-boxes and for supporting user preferences [8].

Another example are reengineering to the Web projects (see [11]) that involve multiple platforms, including legacy platforms, stand-alone desktop applications, and Web application servers. Here, software components must be dynamically configured and composed according to capabilities of the environment into which they are loaded.

If multiple programming languages are used in a system, the configuration and composition aspects are typically handled differently in each language. Cross-language composition is thus hard to understand and maintain. Typical examples that require composition and configuration of components across languages are reengineering projects or software product lines. Here multiple languages need to be supported, for instance, for historic reasons like supporting legacy systems written in other languages than the main system. Composition and configuration concepts should cut across the individual programming languages used. But this is difficult because language concepts are heterogeneous.

Consider for instance the reengineering project in [12] which provides an object-oriented interface and a middleware to a given C system, meaning that C and C++/Java need to be supported. Language integration goes beyond pure invocations in the other language. What is needed are language integration concepts for the different languages' concepts, such as object and type systems, as well as wrapping concepts for (foreign language) components. If the concern “composition and configuration” should be separated in such a system, the composition and configuration concepts must include concepts for language integration as well.

Many composition and configuration approaches only support static composition and configuration (i.e. composition at compile time). Just consider the C++ and Java inheritance relationships, generative programming approaches [13], or aspect-oriented programming in the style of AspectJ [14]. In many scenarios these approaches provide a good means for component composition and configuration; in other scenarios, however, they do not work so well because runtime composition or configuration is inherently needed by the component.

Rapid configuration of server components is another example where runtime behavior composition and configuration is required. 24x7 servers usually should not be shut down for configuration tasks; thus configuration is required while the server is running (see [11]). This problem is partially solved by load-time AOP approaches for application servers, such as JAC [15] or JBoss AOP [16]. However, these approaches can only configure components with behavior while they are loaded into the Java virtual machine – once a class is loaded, the behavior can only be manipulated in predefined ways. Dynamic AOP approaches, such as AspectWerkz [17] or Steamloom [18], even support dynamic re-configuration of the aspects but behavior redefinition of the aspects or aspectized components is not supported (but can be “simulated”).

In summary, the example projects have severe requirements of changing the behavior (or semantics) of their component’s composition or configuration at runtime. Thus, we believe the concern “behavioral composition and configuration” should be supported explicitly.

3 Concept of a tailorable language for configuration and composition support

If we summarize the requirements from the previous sections, we need a configuration and composition concept that:

- supports behavioral definitions of composition relationships and configuration code
- supports behavior (re-)definition at runtime
- is easy to combine with multiple environments (like platforms, one or more languages in which components are written, etc.)
- supports separation of concerns for the concern “composition and configuration”

Our proposed approach to solve these related problems is to define a dynamically interpreted language that is flexibly adaptable (tailorable) to the context in which it is used. Such a language can then be integrated into the languages in which the system’s components are written. All composition and configuration behavior is then externalized into the composition and configuration language.

The concept should be integrated with multiple object-oriented and non-object-oriented languages. In this context a major design obstacle is that the language concepts of those other “host languages” are quite diverse. It should be possible to quickly integrate the object concepts of composition and configuration language with the respective language concepts of the other languages.

The approach we propose builds on an architecture that contains three key elements:

(1) The language chosen should be a *dynamically interpreted language* that is able to evaluate data, provided in the language, as code. Such languages are also called *homoiconic languages* [19]. This language property is illustrated by the simple example in Lisp below, in which we first assign to `a` a program fragment which assigns 1 to `b`. Later we evaluate the program fragment that `a` refers to using `eval`. The consequence is that the code in `a` is executed and `b` is set to 1.
```
(setf a '(setf b 1))
; ...
; some time later
; ...
(eval a)
```
Besides the Lisp family of languages, many other languages are homoiconic, including Tcl, Perl, Prolog, and Smalltalk.

(2) The chosen language should be (or be made) *embeddable* in all other host languages in which the components to be composed and configured are written. “Embeddable” means that the embedded language is used like a library of the embedding language; i.e. the embedding language controls the embedded language. Operations in the embedded language can be invoked from the embedding language, and embedded language can call back into the embedding language. Again, there are many languages supporting this feature. For instance, most scripting language, such as Tcl, Python, and Perl, are embeddable in C/C++ (some of them also in Java). The Java Native Interface supports embedding C programs in Java.

(3) The pattern *Object System Layer* [4] implements an object system as a Layer [20] in another system. This way we can extend systems written in non-object-oriented languages with object-oriented concepts, integrate different object concepts, and extend existing object concepts. In this paper we use the Object System Layer pattern to realize the *tailorability* property of the composition and configuration language.

An example of an Object System Layer architecture is depicted in Figure 1. The business logic components are written in a host language. The host language embeds the composition and configuration language. The composition and configuration language is tailored to the object system of the host language. The Object System Layer can intercept all invocations relevant for composition and configuration. It introduces the behavioral system composition and configuration at these invocations transparently.

Figure 1 should be placed here.

The architecture elements (1) and (2) are relatively easy to realize because existing implementation can be reused. The great number of existing implementations can be used as a guideline for building a "little" homoiconic, embeddable language from scratch, if this is required. The novel aspect of our concept is to combine such a language with an Object System Layer that provides the following properties as extensions of the language:

- *Dynamic object system:* In a dynamic object system all relationships of an object, including class and superclass relationships, might possibly be changed at any time. This especially means that a dynamic object system can be combined with (dynamically loaded) components of the host language at runtime. The important conceptual property is that the object system must contain language constructs that are tailorable to the concepts of the host language. In particular, we want the classes in the Object System Layer to wrap host language classes, and then add the composition and configuration classes to these wrapper classes. We propose two language features to reach this goal. Many other concepts exist that can be used equivalently:
  - *Flexible classification concept:* In our prototype one object can have multiple, dynamic classes, which are composed as mixins [21,22,23]. This feature is used to add the composition and configuration classes dynamically to the host language wrappers. Many other dynamic programming techniques, such as dynamic aspects [17,18], meta-objects [24], or meta-classes [25], can be used equivalently to add behavior to classes dynamically and transparently.
  - *Split objects:* Invocation in the composition and configuration language should be automatically indirected to the wrapped object in the host language. We use the pattern Split Object [26,27] to realize this goal: A split object is an object that exists in two languages, but is treated like a single instance. This way we intercept invocations that are not implemented in the composition and configuration language, and forward them automatically into the host language. An example is shown in Figure 2. To realize this feature we use a *dynamic dispatcher* in the composition and configuration language, and *generative/aspect-oriented programming* in the host language. Again any other technique that is capable of intercepting all invocations to an object can also be used to realize the Split Object pattern (see [26]).
- Reflection: Reflection [28,29] allows to query the runtime state of the object system at any time. We require two kinds of reflective properties in our Object System Layer:
  - *Structural reflection (introspection):* Because we aim at runtime composition, an important goal is to be able to find out the current composition of the objects and classes at any time. Therefore, we need to offer *introspection options* for each language element provided by the Object System Layer. Examples are introspection options for type relationships, superclass relationships, associations, and aggregations. Using introspection options we can find out about the structure of the composed components at runtime. This feature is

used as a “memory” of how the object system is tailored to the structures in the host language. Any other structural reflection technique can be used equivalently.

· *Control flow reflection:* Structure information, as obtained by introspection options, are one part of the current context of an object. The second part is the behavior or control flow context in which the object is invoked. The behavior context is in many interpreted languages handled by the callstack. Thus, the callstack should be fully accessible from within the language. This way we can find out, for instance, which component has called which other component. This feature is used to allow for conditional composition and configuration behavior on basis of the current control flow. Any other techniques that allows for access to the current control flow of the invocations in the Object System Layer can be used equivalently.

We call a language that supports the properties and the architecture sketched above a *tailorable language*. Such a tailorable language can be used to provide the behavioral runtime composition and configuration of components, (possibly) written in other languages. In the remainder of this paper, we show this as follows:

- We explain an excerpt of a case study using this architecture as an example in Section 4
- We provide the details of a prototype implementation in Section 5
- We demonstrate how this prototype can be tailored to the host languages Java, C, and C++ (see Section 6)

Figure 2 should be placed here. †

4 Case study excerpt: Hardware selection for an MHP product line

In this section we present a simple case study of how to apply the concepts from the previous section in a typical, practical composition and configuration situation (see [8] for more details). The sample code excerpts are written in our prototype implementation Frag, explained in Section 5.

Consider you are developing a product line of software components for the Multimedia Home Platform (MHP) [10]. The MHP specification [10] is a generic set of APIs for digital content broadcast applications, interaction via a return channel, and internet access on an MHP terminal. Typical MHP terminals are for instance digital set-top boxes, integrated digital TV sets, and multimedia PCs. The MHP standard defines a client-side software layer for MHP terminal implementations, including the platform architecture, an embedded Java virtual machine implementation running DVB-J applications, broadcast channel protocols, interaction channel protocols, content formats, application management, security aspects, a graphics model, and a

GUI framework.

Even though there are just a few MHP set-top boxes available right now, a great diversity of products can be foreseen in this field. In addition, there will be a great number of supporting systems, such as input devices. The whole variety of different TV systems and PCs also might be used as an MHP client platform. An MHP product line needs support dynamic configuration of the hardware during application startup on the client platform.

Software products that should run on a MHP platform need to consider multiple kinds of variation regarding the hardware environment. As MHP applications are broadcasted, the earliest possible binding time for the variation points is during startup of the application on the client platform.

Let us consider a few typical hardware selection problems. Different input devices offer various input events, and the MHP application needs to dynamically configure itself to these hardware elements. Graphical elements in the MHP API include a few graphic functions of Java, as well as the more advanced GUI elements of the HAVI standard [30]. It depends on the type of set-top box if HAVI is supported or GUI elements have to be painted using the graphic functions of Java. Different back-channels can be identified at runtime and may in some situations be handled differently.

In pure Java these issues are typically handled by loose associations with objects. This, however, does not provide a central variation point management architecture that, for instance, supports querying various hardware options or reconfiguring them at runtime with a common model.

The solution idea, proposed in this section, is to use our prototype Frag as a composition and configuration language for hardware-related software components. We use scripts like the following example for behavioral compositions and configurations:

```
...
MHPButton create OK -set parent OKBOX
MHPButton create CANCEL -set parent OKBOX
if {$earlyVisible == 1} {
  CANCEL setVisible
  OK setVisible
}
...
```

In the script two button components are created and composed with another component `OKBOX`. Then the components are configured conditionally: If the variable `earlyVisible` is true, the configuration option `visible` is set on both components.

Scripts, like this example, can be rapidly composed and changed, and they can be stored in a central repository. In Java, the classes that implement the respective GUI components (such as the `MHPButton` in the example above) support a common interface. The common interface is mapped to the HAVI API or the Java-based Graphics API respectively:

```
class HAVI_Button implements MHP_Button {
  HtextButton button;
  // use HAVI HtextButton to draw a button
  ...
}
class Graphics_Button implements MHP_Button {
  // use Graphics API to draw a button
  ...
}
```

These classes are bound to Frag classes using the pre-defined class `JavaClass`. All classes derived from `JavaClass` are wrappers for classes that are defined in Java. That is, all instances of the classes are Split Objects between Frag and Java.

We can select between the HAVI implementation and the Graphics implementation by using the following behavioral composition script:

```
if {[hardwareSelection haviSupport] == 1}
  ...
  JavaClass create MHPButton -set javaClass HAVI_ButtonCmd
  ...
} else {
  ...
  JavaClass create MHPButton -set javaClass Graphics_ButtonCmd
  ...
}
```

The resulting object structure of the example, if HAVI is selected, is shown in Figure 3. †

Figure 3 should be placed here.

Let us compare the solution briefly to composition and configuration using hard-coded invocations in Java. The solution has many benefits: Behavioral composition and configuration is not tangled in the business logic code. Thus the solution supports better understandability and maintainability. The developer did not have to care at all for language integration: This was done automatically by using `JavaClass` in Frag (and the Split Object pattern hidden by this class). Thus the solution is easy to use. As Frag is dynamically interpreted the composition and configuration behavior can be changed at runtime. Thus the solution supports unanticipated software evolution. There are a few disadvantages as well: Each composition/configuration object consumes memory, and invocations routed through two languages are slower then invocation performed in only one language. This is not a big drawback, as it only applies for composition and configuration code, which is not a large part of the system and does not affect performance-critical parts of a system. A second drawback is that there is more learning effort required: Developers must learn the Java MHP system plus the configuration options and syntax of Frag. Overall, the benefits clearly outweigh the drawbacks in this example.

5 Details of the prototype implementation Frag

We have realized the concepts, introduced in the previous sections, in a prototype implementation called Frag [31]. Frag extends the existing scripting language Tcl [32] with an Object System Layer that supports the properties described in the previous sections. We have chosen Tcl because it is easily embeddable in C/C++ and Java, which are languages typically used in our projects, and because Tcl is also a homoiconic language (i.e. code can be provided and evaluated at runtime). Many other languages have similar properties and could have been chosen as well, such as Lisp, Smalltalk, Ruby, Python, or Perl.

In the following sections, we describe some of the details of the Frag implementation, because we think they are not obvious and important for realizing the concepts in other environments. Note that even though all conceptual ingredients of the solution must be present, there are many alternative realization solutions possible.

Frag is developed as an open-source project and can be downloaded from [31]. In this paper we describe only the concepts of our prototype implementation as needed for the paper. For further details, a language reference describing the syntax and semantics of Frag can be found online at [33]. Frag requires either Tcl [32] (to be run as a C/C++ extension) or Jacl [34] (to be run as a Java extension). Jacl is an implementation of a Tcl interpreter in plain Java.

5.1 Dynamic and Tailorable object and class concept

In Frag, every entity is represented by an object. 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 the host language. The goal of this tailorable class concept is to support many different language concepts.

For instance, consider inheritance. Frag is capable to integrate different inheritance models with one class concept, including static multiple inheritance as in C++, static single inheritance and interfaces as in Java, dynamic inheritance as in XOTcl [35], mixin-based inheritance [21,22], and procedural concepts simulating inheritance (as sometimes used in C, Tcl, or Cobol). This concept is different to object-based languages, such as Self [36], that do not language-support inheritance and use delegation to model inheritance instead.

Let us briefly introduce the dynamic object and class concept of Frag with some examples. In Frag an object is created by invoking the method `create` of its class. For instance, we can create an object `Connection1` from the most general class `Object`:

```
Object create Connection1
```

This invocation means that a new object with the object ID `Connection1` is created. This new object has the class `Object`.

Each object can have one or more classes, but each object in Frag has at least one class. An object always has at least the class `Object`, the most generic class in the Frag object system. Each other class is a subclass of `Object`. This way it is ensured that every object in Frag can use the features defined for the class `Object`.

We have already seen that the method `create` can be used to derive an object from a class. Consider we want to create a class `SocketConnection`, and derive an object `Connection2` from this class. This is done as follows:

```
Object create SocketConnection
SocketConnection create Connection2
```

In the context of `Connection2`, the object `SocketConnection` acts as a class. Note that there is no difference in how we have created `SocketConnection` and `Connection1`; at this point both are ordinary objects. The subsequent creation of `Connection2` is what makes `SocketConnection` (also) a class. In other words, `SocketConnection` acts as a class only in the context of `Connection2` (and its other instances).

In this example it does not make much sense to derive objects from `Connection2`; thus it will likely not act as a class. But note that this two-level class concept is not always applying. For instance, we might want to constrain or extend the features of `SocketConnection`.

Just consider a typical situation in a composition and configuration language like Frag: Frag does not implement the `SocketConnection` itself, but it is a wrapper for a Java class. That means somewhere we need to define how to access the Java counterpart of `SocketConnection`, and this implementation should be reusable for all Java classes. This can be done by giving the `SocketConnection` itself a class `JavaClass` which can be used for other Java classes as well:

```
Object create JavaClass
JavaClass create SocketConnection
SocketConnection create Connection3
```

All three example `classes` relationships are depicted in Figure 4.

Classes of classes are used to configure the split objects, as shown in the example in Section 4 (where we have used `JavaClass` already).

Note that a number of inheritance concepts, such as Smalltalk [37], XOTcl [35], or SOM [25], see the class of a class as a so-called *meta-class* (such as `JavaClass` in the example above). When an object system distinguishes classes and meta-classes explicitly, there are a number of open issues, including the number of meta-levels and how to compose meta-classes with meta-classes [25]. Frag avoids these problems by making all classes ordinary objects and interpreting their own class relations only in their particular context. In Section 5.3 we explain how these classes are unambiguously composed using a linearized method resolution order.

Figure 4 should be placed here. †

In Frag, each object can have multiple classes. This feature can be used, for instance, to implement the pattern Decorator [38]. Consider a simple example: we want to log the invocations of `Connection3` using the class `ConnectionLogger`. This class can be added to `Connection3` by adding it in addition to the existing class `SocketConnection` using the method `classes`:

```
Connection3 classes {ConnectionLogger SocketConnection}
```

Now `Connection3` has two classes, `ConnectionLogger` and `SocketConnection`. Note that we still have to compose the two classes properly to make `ConnectionLogger` a Decorator of `SocketConnection`. This functionality is provided by the language element `next`, explained in Section 5.3. `classes` can be used to dynamically change the classes of an object at any time.

The feature “multiple classes” is used to dynamically add behavior to the split objects at runtime.

In Frag each class can have one or more superclasses. Superclasses are defined for a class with the method `superclasses`. Per default, each class has the superclass `Object`. This class does not have to be explicitly provided as a superclass. For instance, if we want to provide a class for logged connections, we can provide the two classes `ConnectionLogger` and `SocketConnection` as superclasses:

```
Object create LoggedSocketConnection \
  -superclass {ConnectionLogger SocketConnection}
```

Note that the superclass relationship is also dynamic. Again this feature is used to tailor the object system to a specific context. For instance, in the `JavaClass` example above, the constructor of `JavaClass` dynamically registers a superclass `Java` for each of its instances that actually implements the reflective connection to the Java language. In other words “dynamic superclasses” can be used to dynamically tailor the behavior of a class and all its instances.

5.2 Dynamic methods

Frag supports dynamic methods. That is, methods can be defined, redefined, and deleted during runtime.

A method is simply defined by invoking the `method` operation and providing it with the method name, the parameter list, and the body of the method.

```
ConnectionLogger method writeMsg {channel msg} {
  puts $channel $msg
}
```

All parameters passed to Frag methods are strings. Thus the parameters `channel` and `msg` in the example method above have no type definitions in their signature, just parameter names. Wrappers to statically typed host language objects need to take care for type conversions (see Section 6). 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 for easily integrating various host languages.

The “dynamic methods” feature is used to define new behavior at runtime.

5.3 Mixin methods and the method resolution order

The class hierarchy is determined by the class and superclass relationships of an object. The result is a directed, acyclic graph. This graph is interpreted in the specific context of each object. The class and superclass relationship graph might contain more than one way to reach one and the same class from one object. To avoid ambiguities, Frag uses a simple resolution rule to determine the order of dispatch: each class of an object is traversed one after another. For each class all of its superclasses are traversed. For each class first the whole superclass tree is traversed completely, before the next class (and its superclasses) are traversed. This way a full list of all classes applicable for an object is created recursively. This list is linearized to contain each class exactly once: always the last occurrence of a class in the list is chosen. This way it is ensured that `Object` is always the last class in the list, because it is the superclass of all other classes.

In the description so far, the most specific class in the hierarchy overrides the more general classes that come later in the linearized method resolution order. That is, if two classes define a method with the same name, the method of the more specific class is executed. Sometimes we want this behavior, but especially in a component composition context there are other cases where we want to compose two methods as in the Decorator design pattern [38].

Just consider again the `ConnectionLogger` class for socket connections from the example above. Here, we want to observe specific methods of the socket connection (like `open`, `close`, etc.), but the logger should not override the methods of `SocketConnection`.

To resolve this problem, Frag offers a primitive `next` to proceed in the linearized order of classes from within a method. Consider the method `close` should be logged before and after a connection is actually closed:

```
ConnectionLogger method close {} {
  self writeMsg stdout "Before close [self]"
  next
  self writeMsg stdout "After close [self]"
}
```

The `next` primitive in between the two output statements forwards the invocation to the next class(es) on the linearized method resolution order (also called “next path”). That is, `close` is searched on the subsequent classes on the next path and invoked, if found.

Note that this behavior is very important for wrapping existing classes. Without it we would have to make changes to the wrapped class or its client in order to introduce a Decorator, something that we often cannot do, because there is no access to these classes. Note that composition with `next` provides language support for the pattern Decorator.

Using `next` a Frag method can be used as a *mixin* method. Frag’s dynamic class concept, supporting multiple classes and multiple superclasses, together with mixin methods, can be used to realize *dynamic mixin-based inheritance*. The mixin classes can be used as small units of composition that are not necessarily defined completely, and that can be mixed into the class hierarchy at arbitrary places, as first envisioned in Flavors [21]. Consider the class `ConnectionLogger`: it can be mixed onto a single object, but also onto a class implementing the `Connection` interface. This way Frag realizes the concept of mixin-based inheritance [22,23] in a dynamic fashion.

Figure 5 shows the full class and superclass graph for `Connection3` on the left hand side. The class and superclass graph from the perspective of `Connection3` (on the right hand side) is much simpler, because all relationships that are not relevant for the dispatch in the context of `Connection3` are removed. From the few remaining relationships duplicates are eliminated, so that the next path order, as depicted in the figure, results.

†

Figure 5 should be placed here.

5.4 Dispatcher

A very important feature of Frag for the integration with other host languages is the `dispatcher` method. We have already explained that Frag first tries to resolve a method within the class hierarchy of the object. A method, however, which is really implemented in the host language cannot be found in the Frag class hierarchy, unless we write a wrapper method for each host language method. In Java, for instance, we could look up such methods using reflection. To do so, we need some way to invoke this lookup process and the method invocation.

Such tasks can be handled by the special method `dispatcher` that might be defined for each class. When a method is not found on the class hierarchy, Frag does not raise an error immediately, but invokes the method `dispatcher`. The dispatcher method of the top-most class `Object` does per default nothing else than raise the error "Method not found". In other words, if `dispatcher` is not overridden by a subclass, an unknown method invocation causes a runtime error.

Subclasses can override `dispatcher` and thus handle messages, not defined in Frag, arbitrarily. For instance, the method for dispatching to Java (explained in detail in Section 6) implements a dispatcher that invokes the Java reflection API to look up the method.

In Figure 6, a method `close` is dispatched for a class `SocketConnection`. As it cannot be found in the class hierarchy, `dispatcher` is invoked with the argument `close`. `dispatcher` is overridden in the superclass `Java` of `SocketConnection`. Here, it is defined that a Java proxy is invoked that is of the Java class `ReflectObject`. This class looks up its arguments via reflection and invokes the Java method if it is found. Note that we have omitted type conversions and lookup invocations to simplify the figure.

†

Figure 6 should be placed here.

5.5 Structural reflection using introspection options

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 offers 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, summarized in Table 1. Subclasses can extended `info` with additional options.

†

Table 1 should be placed here.

Let us briefly explain one example to show how `info` works. Consider you want to add a `ConnectionLogger` to an object as a Decorator, as introduced above. But in contrast to the simple example above, you do not know the list of classes. This is a typical situation in a wrapping context because other, independent extensions might have introduced Decorators before. With the introspection option `classes`, we can add the Decorator to the current class list by concatenating the `ConnectionLogger` and the current classes:

```
Connection3 classes [concat ConnectionLogger [Connection3 info classes]]
```

In other words, we can introduce Decorators transparently, without making any assumptions about other extensions of the class.

5.6 *Control flow reflection using callstack information*

In Frag the control flow is handled on Frag's callstack. This callstack is fully accessible from within the language.

All callstack information is provided using the object `callstack`. The keyword `self`, used in earlier examples, is actually a shortcut to `callstack self`, which returns the top-level object on the callstack. With the callstack information options, summarized in Table 2, objects, methods, and classes at any level of the callstack can be queried. In typical programming situations, the top-level (level 0) and the calling level (level 1) that represent the current and calling scope are important.

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

```
X method callerPrinter {} {
  puts "Invoker: [callstack callingObject]->[callstack callingMethod]"
}
```

The feature "control flow reflection" can be used, for instance, to make conditional composition or configuration decision, based on the object/class/method that has invoked a method.

†

Table 2 should be placed here.

6 Integration with host languages

In this section, we explain the integration of Frag with the host languages Java, C, and C++. For integrating these languages, we use the split object concept and Frag's language concepts, introduced in the previous sections. In other words, we give examples of how to tailor the language Frag to the concepts of other languages (Java, C, and C++).

6.1 *Tailoring Frag to Java*

Integration with a host language is typically done by providing a component that tailors the Frag object system to the particularities of the host object system. In case of Java, the integration component consists of two main classes: `JavaClass` and `Java`.

The class `JavaClass` is used for wrapping a Java class with a Frag object. Any instance of this class is a split object; and invocations are automatically forwarded into Java and vice versa. Both, wrapping a Java class and forwarding invocations, is implemented using Java reflection. Internally primitive Java types (like int, boolean, etc.) are automatically converted to and from strings. Non-primitive Java types have to be wrapped in order to be used from Frag.

Figure 7 shows the structure of the split object solution. A split object class in Frag is derived from the class `JavaClass` which overloads the standard creation process of Frag: it does not only create a Frag instance, but also a Java instance. To do so, `JavaClass` looks up the respective Java class via Java reflection. All split objects automatically get the superclass `Java` which overloads the standard dispatcher. This dispatcher forwards the invocation to the Java instance, if the operation cannot be found for the Frag instance.

†

Figure 7 should be placed here.

Some language semantics of the Java language must be provided by the split object because they cannot be ignored. Even though type conversion is handled automatically on the class `Java`, sometimes we need to care for type casting. For instance, when an object is obtained from a Java hash-table, the result is returned as a `java.lang.Object` – and not as the "real" type put into the hash-table. Such re-typing to more generic types, performed inside of Java classes, cannot be deduced by the automatic type converter. The class `Java` contains a method `cast` for handling type casting.

If we dispose a split object in Frag, the Java garbage collector automatically frees the Java instance as well. The class `Java` ensures that a Java reference is hold until the Frag destructor is called – this way the correct destruction order is ensured automatically.

All other host language particularities of Java are handled in the same way – by implementing them on the two classes `JavaClass` and `Java`. These classes tailor all derived classes so that they do not violate semantics of the host language.

6.2 Automatically accessing Frag from Java

We also need to access Frag objects from Java. To do so, we have to instantiate a Frag interpreter from Java and then we can perform invocations using the method `eval`. We can insert such invocations, wherever we need them, because Frag is *embedded* in Java.

Sometimes this is not enough: consider we want to automate the indirection of certain invocations in the host language into the composition language. For instance, all instances of the Java class `Object1` might require configuration by split objects. It is cumbersome to insert `eval` invocations around each Java invocation.

A better solution is to automate the indirection. Luckily, there are good tools available to realize such an indirection, namely program generators and AOP tools. In our prototype implementation we use AspectJ [14]. Many other AOP tools would be able do the same job.

For all specified invocations, our AspectJ aspect invokes Frag objects instead of the Java invocations. Invocations to classes that are instrumented in this way have to go through Frag before they reach the original receiver.

Essentially, the AspectJ aspect automates the concept that we have already depicted in Figure 2. The client "virtually" wants to call `Object1` in Java. But "really" the invocation is intercepted, sent to the split object for `Object1`, and then `Object1` in Java is really invoked. The split object thus can do composition or configuration tasks in a dynamic fashion without the Java object noticing it.

6.3 Integration of C and C++

The concepts described for Java work in the same way for other languages, such as C and C++.

Integration of Frag with C and C++ is quite similar to integration with Java, except that these languages do not provide a reflection API. Thus it is not possible to dynamically look up C++ classes and methods or C structs and functions. The standard Tcl approach for integration is to use function pointers and write the wrappers by hand or with a code generator. A code generator that supports Tcl wrapper generation for C and C++ is SWIG [39].

C functions and C++ methods are wrapped using function pointers. In C we usually try to align some structures and functions that semantically belong together to form a split object. In C++ we additionally have to map the C++ object identity to Frag object IDs. Typically we additionally mimic relevant parts of the C++ class hierarchy in Frag and let these class inherit from a class `C++`, the superclass for all C++ split objects. This class handles the integration with the C++ class concept. For instance, it invokes `new` or `destroy` in C++ when the respective constructor or destructor are invoked in Frag. As C and C++ contain no garbage collector, it is important to care for the destruction order of split objects so that no memory leaks occur or memory is not freed twice.

7 Trade-off evaluation

In Sections 1 and 2 we have provided a motivation of the problems that might lead to the adoption of our approach: Behavioral composition and configuration is needed in an easy-to-use manner, perhaps even at runtime, and/or the concerns "composition and configuration" should be separated from the business logic. As described in Section 2.2, many projects have such requirements. To avoid the unnecessary complexity of an additional software framework, of course, the approach should only be used, if these requirements occur in a project. If this is the case, there are, however, other solutions for composition and configuration. For instance, in Section 1 we have given the example of composition and configuration in the EJB framework and mentioned simple hard-coding as another alternative solution. In this section we want to evaluate the factors that are drawbacks of our approach or might be conceived as problems. The goal is to allow an architect or developer to make an informed trade-off analysis between our approach and alternative approaches.

A drawback of our approach is that a second language has to be learned by developers, as well as the additional APIs of the composition and configuration language implementation. This learning effort has to be contrasted to that of other solutions. For instance, in the EJB example in Section 1 the EJB APIs and XML deployment descriptor language has to be learned.

The split object approach has the drawback that each invocation has to be dispatched twice, once in the host language and once in the composition and configuration language. This means that split object invocations take at least twice as long as ordinary invocations. This estimation is not necessarily correct, however, when the performance of the whole application is compared: The composition and configuration code performed in the composition and configuration language would be needed anyway, so it is not clear that the split object implementation is really slower than a composition and configuration programmed in the host language. Thus what actually needs to be compared is the performance of the composition and configuration language and the host language performance.

A similar trade-off analysis can be made for other quality attributes [3], such as memory consumption, program reliability, and productivity. It is hard to give a general estimation for these factors, however, because the quality attributes *are not independent* of each other and they are not independent of the quality attributes that our approach aims at, i.e. maintainability, understandability, reusability, and flexibility. A general trade-off analysis is thus not possible because the quality attributes of an architecture can only be judged using the *domain-specific* requirements of that particular architecture. Finally, evaluations of these attributes are always *language-dependent* and *implementation-dependent*. Our prototype implementation, Frag, however, is only one possible implementation variant of the concepts. The concepts can be implemented in many other ways – with quite different impacts on the quality attributes mentioned above.

Having said that, for Frag or similar implementations (that is: implementations reusing a scripting language) we can make some estimations for the trade-offs compared to a hard-coded native C/C++ or Java implementation. Note that these results are observed in our examples and case studies and cannot necessarily be generalized. Also note that Prechelt comes to similar results in an empirical comparison of these and other factors in a number of languages including Tcl, C, C++, and Java [40]. Designing and writing the program in scripting languages such as Tcl takes no more than half as much time as writing it in C, C++, or Java, and the resulting program is only half as long. The typical memory consumption of a script program is about twice that of a C or C++ program. For Java it is another factor of two higher. C/C++ programs are always significantly faster than Java and script programs. In the majority of cases, Java programs are faster than the script programs, in other cases the script programs outperform Java programs. No unambiguous differences in program reliability between the language groups can be observed. In our implementation approach an additional memory consumption penalty for the memory used by the script interpreter should be considered. Again, this must be contrasted to the memory consumption of alternative solutions for component composition or configuration

Because we use a two-language-approach, the drawbacks of one language implementation can compensated in the other language. For instance, performance-critical code can be moved from the composition and configuration language into the host language, whereas performance-uncritical code is placed in the composition and configuration language to benefit from the productivity gain of that language.

Our conclusion from these results is that the Frag prototype implementation and similar implementations of our concepts have a moderate performance and memory consumption penalty, but offer benefits in terms of productivity and program length. For performance-critical and memory-critical applications, for instance embedded systems, the performance and memory consumption penalty might be a problem, for many other application areas this should be tolerable. A final estimation, whether our approach is appropriate or not cannot be provided in general, but must be made domain-specifically, because other factors, such maintainability, understandability, reusability, and flexibility, must also be considered.

8 Related work

8.1 *Frameworks for composition and configuration of software components*

Current mainstream server component frameworks, such as Enterprise Java Beans (EJB), the CORBA Component Model (CCM), or COM+, mainly provide black-box components [5,41]. Both approaches compose components using a component container: usually glue code can be generated for given components, which handles the composition, and annotations (XML metadata) are used for configuration. These approaches have had much industrial success and in them configuration and composition concerns are treated as first-class entities of the software architecture. However, in contrast to our approach, unforeseen behavioral configuration and composition requires hard-coding. Runtime configuration and composition is only supported in a declarative manner. Our approach can be used to extend existing server component frameworks with support for behavioral runtime composition and configuration.

Jiazzi [42] is a component infrastructure for Java. It does not primarily describe the connectors and flows between architectural elements, but the "uses" relationships. ArchJava [43] is also a Java extension that embeds the architectural knowledge and the code in the same document. These approaches have the general goal to make component composition explicit, a commonality with our approach, but use a different solution: architectural elements are described in a programming language extension that resembles an architecture definition language. Both approaches can untangle the concern "composition", but do neither provide support for behavioral composition nor runtime composition.

The VCF framework [44] integrates components from various Java component models, namely COM+, Corba, EJBs, and Java Beans. A Facade component is provided that can load plug-ins for the various component technologies. The Facade provides a common denominator interface of the technologies to clients. VCF has a different architecture than our approach because it focuses more on integration. Thus VCF has a richer model of pre-defined integration behavior, but VCF does not provide for user-defined extensibility of the composition/configuration behavior as our approach does.

8.2 *Languages for composition and configuration of software components*

Other popular component approaches are component frameworks [45] introduced by scripting languages like Tcl, Perl, Ruby, or Python. A component framework is a collection of software components and architectural styles that determine the interfaces that components may have and the rules governing their composition [46].

As a Tcl extension, our prototype Frag builds on the experiences gathered in those languages. However, our concepts extend those of the other object-oriented scripting languages, such as Ruby, Python, Perl, and XOTcl [35]. These languages are implemented in C and thus hard to embedded in other host languages than C/C++ such as Java. Also, our concepts provide a simpler and more flexibly tailorable language. For instance, the dynamic mixin-based inheritance feature allows for simple Decorator-style composition of split object classes and extensions. All named scripting languages can nonetheless be used as a starting point for implementing the Object System Layer required in our concepts, and are important predecessors of our approach.

Lua [47] is an embeddable programming language library similar to other scripting languages. In contrast to most other scripting languages, however, Lua offers support for meta-programming and reflection that is intended to be used for creating domain-specific languages. This makes Lua similar to our approach, because it is tailorable in the sense that it provides a language framework for creating other languages. Thus it should be easier to create a composition and configuration language in Lua, and to adapt Lua to other host languages, than it is in many other languages. What is missing in Lua to realize our concepts is a construct to untangle the composition and configuration concerns, such as dynamic mixin-based inheritance or dynamic aspects.

A number of more dynamic, object-oriented environments, such as CLOS [48], Smalltalk [37], and Self [36], provide both a programming environment and a program execution environment, allowing one to influence the language behavior from within a program. Our approach utilizes this important concept of these languages.

Introspection options are a simplified form of the language concept of computational reflection [28,29]. The dynamic, tailorable object system aims to provide a similar expression power as meta-object protocols (MOP) [24], but in a more easy-to-understand and easy-to-use manner. Besides these simplifications, Frag extends these languages' concepts with language extensibility and embeddability concepts.

Piccola [49] is a language for building applications from software components. The semantics of Piccola is defined in terms of a process calculus in which values are communicated as nested records, called forms. Piccola is more "pure" than our approach in the sense that it only provides compositional features as language elements. A consequence of this strong focus is, however, that arbitrary behavioral composition is not possible – which is a major goal of our approach.

8.3 Mixin concepts

Classes are not always suitable as a unit of composition: On the one hand, a unit of reuse should be small and flexibly composable with arbitrary kinds of other classes. On the other hand, an object's type needs to defined completely and requires a fixed place in the class hierarchy. Moon proposed flavors as an early attempt to solve this problem [21]. Here, a flavor is a small unit of composition that is not necessarily defined completely. It can be *mixed* into a given class hierarchy at arbitrary places.

Moon's approach is refined by the concept of mixin-based inheritance [22,23]. In this approach, a mixin is a class whose superclass is not specified at mixin implementation time, but is left to be specified at mixin use time. The advantage is that a single mixin can be used as a subclass for multiple other classes. Thus, mixins can be reused across a number of base classes. However, still this approach suffers from the problem that the mixins are derived by inheritance. Therefore, it is hard to write generic, reusable composition units using this approach alone.

The mixin concept has a strong influence on the object system design in our work. In order to use mixins for dynamic component composition, we had to extend the mixin concept with a concept for dynamic mixin composition (dynamic mixin-based inheritance) and a concept for behavior redefinition in the mixin classes.

8.4 Aspect-oriented component composition

Aspect-oriented programming (AOP) [50] approaches can be used to untangle composition- or configuration-related concerns from a software system, and define them in aspects.

AspectJ [14] allows to declaratively provide "pointcuts" that specify "joinpoints". Joinpoints are specific, well-defined events in the control flow of the executed program. At these joinpoints "advices" can be introduced, which define new behavior to be added before, after, or around the joinpoint. Aspects are cleanly separated (untangled) from the business logic code. Thus AspectJ can be used for static behavior adaptation, a feature that we use for language integration with Java (see Section 6). However, as AspectJ uses a compile-time tool, called the aspect weaver, for instrumentation, it does not support the main goal of our approach: dynamic behavior redefinition for composition and configuration concerns.

JAC [15] is a framework for dynamic, distributed aspect (server) components. As in Frag and in contrast to AspectJ [14], methods in focus of structural or behavioral aspects are specified with strings and looked up reflectively. JAC's dynamic wrappers are method interceptors, similar to Frag's mixin methods. JAC allows to configure aspects using domain-specific languages. The elements of the language are corresponding to method names of the aspect component. This is similar to how Java/C/C++ elements are bound to Tcl/Jacl in Frag. JAC uses load-time instrumentation to introduce aspects, whereas Frag uses runtime dynamics.

Other dynamic AOP approaches in Java are AspectWerkz [17] or Steamloom [18]. Here, "dynamic" does not mean that behavior can be redefined at runtime. Just the aspect composition of "expected" classes can be changed. Even though behavioral composition or configuration is much less convenient than in our approach (behavior changes require re-loading and re-weaving), it is possible to use these tools for behavioral composition or configuration. The benefit of using these approaches, compared to our approach, would be that the developer could work in one language (here: Java) only, and does not have to learn multiple languages.

AspectS [51] provides a runtime aspect weaver for Smalltalk. It uses a message interceptor concept, called method wrappers, and adds them to the program using meta-programming techniques. In particular, method wrappers allow for the introduction of code that is executed before, after, or around an existing method. Method wrappers replace an entry in a class method dictionary, add behavior to the method invocation, and eventually invoke the wrapped method itself. Frag implements a similar basic concept in its mixin methods, but extends the concepts in various ways – especially we have extended the mixin concept with dynamic mixin-based inheritance and extended the language with tailorability concepts. AspectS/Smalltalk do not focus on embeddability, but because all other elements of our approach are supported in some way, we believe it is quite straightforward to realize our concepts in AspectS for Smalltalk components.

9 Conclusions and further work

Behavioral composition and configuration of software components is an important issue in many software systems. Unfortunately, the support for this concern is not a first-class citizen in many component models, especially if behavior changes are required while the system runs. We have pointed out that there are many concepts that solve parts of this problem: scripting languages introduce embeddability in other languages, homoiconic languages allow for behavior redefinition at runtime, mixin approaches allow for extending classes transparently, patterns like Object System Layer and Split Object provide language concept extensions and integration, reflection supports informed decision about the system's structure and control flow, and so forth.

The contribution of this paper is to provide a language-based approach and architecture that integrates these important related approaches into a single concept for behavioral composition and configuration of software components at runtime. This concept allows for untangling composition and configuration concerns from the business logic code – into mixin classes defined in the composition and configuration language. The composition and configuration language is a tailorable language, meaning that it can easily be adapted to new languages and new component models.

Even though each of the ingredients is quite complex on its own, we believe that the combination is nonetheless useful for two reasons: (1) as shown in the example in Section 4 the concepts are quite easy to use once they are implemented, and (2) the implementation can be based on existing technologies that hide the complexity. We have demonstrate this using our prototype, Frag, which reuses the Tcl language, Java reflection, AspectJ, and SWIG.

As further work we plan to provide more integration models than just Java and C++ for the prototype, and extend the concepts to support dynamic AOP languages. We also want to extend the concepts to support distributed components provided as Web services.

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. V. Lopes, D: A language framework for distributed programming, Ph.D. thesis, College of Computer Science, Northeastern University (Dec 1997).

[3] L. Bass, P. Clement, R. Kazman, Software Architecture in Practice, Addison-Wesley, Reading, USA, 1998.

[4] M. Goedicke, G. Neumann, U. Zdun, Object system layer, in: Proceedings of 5th European Conference on Pattern Languages of Programs (EuroPlop 2000), Irsee, Germany, 2000, pp. 397–410.

[5] C. Szyperski, Component Software – Beyond Object-Oriented Programming, ACM Press Books, Addison-Wesley, 1997.

[6] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach, Addison-Wesley, 2000.

[7] M. Völter, A taxonomy for components, Journal of Object Technology 2 (4) (2003) 119–125.

[8] M. Goedicke, C. Koellmann, U. Zdun, Designing runtime variation points in product line architectures: three cases, Science of Computer Programming 53 (3) (2004) 353–380.

[9] M. Goedicke, K. P. und Uwe Zdun, Domain-specific runtime variability in product line architectures, in: Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS 2002), LNCS 2425, Springer Verlag, Montpellier, France, 2002, pp. 384–396.

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

[11] U. Zdun, Reengineering to the web: Towards a reference architecture, in: Proceedings of Sixth European Conference on Software Maintenance and Reengineering (CSMR'02), Budapest, Hungary, 2002, pp. 164–176.

[12] M. Goedicke, U. Zdun, Piecemeal legacy migrating with an architectural pattern language: A case study, Journal of Software Maintenance and Evolution: Research and Practice 14 (1) (2002) 1–30.

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

[14] 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.

[15] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, JAC: a flexible framework for AOP in Java, in: Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, 2001, pp. 1–24.

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

[17] J. Boner, A. Vasseur, AspectWerkz, http://aspectwerkz.codehaus.org (2004).

[18] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann, Virtual Machine Support for Dynamic Join Points, in: Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD 2004), ACM Press, Lancaster, UK, 2004, pp. 83–92.

[19] A. Kay, The reactive engine, Ph.D. thesis, University of Utah (1969).

[20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-orinented Software Architecture - A System of Patterns, J. Wiley and Sons Ltd., 1996.

[21] 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.

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

[23] G. Bracha, G. Lindstrom, Modularity meets inheritance, in: Proceedings of the IEEE Computer Society International Conference on Computer Languages, IEEE Computer Society, Washington, DC, 1992, pp. 282–290.

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

[25] I. R. Forman, S. H. Danforth, Putting Metaclasses to Work – A new Dimension to Object-Oriented Programming, Addison-Wesley, 1999.

[26] U. Zdun, Some patterns of component and language integration, in: Proceedings of 9th European Conference on Pattern Languages of Programs (EuroPlop 2004), Irsee, Germany, 2004, pp. 1–26.

[27] U. Zdun, Using split objects for maintenance and reengineering tasks, in: 8th European Conference on Software Maintenance and Reengineering (CSMR’04), Tampere, Finland, 2004, pp. 105–114.

[28] 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.

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

[30] HAVI, HAVI specification 1.1, http://www.havi.org (May 2001).

[31] U. Zdun, Frag, http://frag.sourceforge.net (2005).

[32] J. K. Ousterhout, Tcl and Tk, Addison-Wesley, 1994.

[33] U. Zdun, Frag language reference, http://frag.sourceforge.net/doc/language.html (2005).

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

[35] 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.

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

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

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

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

[40] L. Prechelt, An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl, IEEE Computer 33 (10) (2000) 23–29.

[41] M. Völter, A. Schmid, E. Wolff, Server Component Patterns – Component Infrastructures illustrated with EJB, J. Wiley and Sons Ltd., 2002.

[42] S. McDirmid, M. Flatt, W. C. Hsieh, Jiazzi: New-age components for old-fashioned java, in: Proceedings of the Object Oriented Programming Systems, Languages, and Applications, Tampa, Florida, USA, 2001, pp. 211–222.

[43] J. Aldrich, C. Chambers, D. Notkin, Archjava: Connecting software architecture to implementation, in: Proceedings of the International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA, 2002, pp. 187–197.

[44] J. Oberleitner, T. Gschwind, M. Jazayeri, Vienna component framework enabling composition across component models, in: Proceedings of the International Conference on Software Engineering (ICSE 2003), Portland, Oregon, USA, 2003, pp. 25 –35.

[45] J. K. Ousterhout, Scripting: Higher level programming for the 21st century, IEEE Computer 31 (3) (1998) 23–30.

[46] J.-G. Schneider, O. Nierstrasz, Components, scripts and glue, in: L. Barroca, J. Hall, P. Hall (Eds.), Software Architectures – Advances and Applications, Springer, 1999, pp. 13–25.

[47] R. Ierusalimschy, Programming in Lua, Lua.org, Ingram (US), Bertram Books (UK), 2003.

[48] 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) (1988) 1 –142.

[49] F. Achermann, O. Nierstrasz, Applications = components + scripts, in: M. Aksit (Ed.), Software Architectures and Component Technology, Kluwer, 2001, pp. 261–292.

[50] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings European Conference on Object-Oriented Programming (ECOOP'97), LCNS 1241, Springer-Verlag, Finnland, 1997, pp. 220–242.

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

Biographical sketch

Dr. Uwe Zdun is working currently as an assistant professor in the Department of Information Systems at the Vienna University of Economics and Business Administration. He received his Doctoral degree from the University of Essen in 2002. His research interests include software patterns, software architecture, scripting, object-orientation, AOP, and Web engineering.

Uwe has published in numerous conferences and journals, and is co-author of the book "Remoting Patterns – Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware" published by J. Wiley & Sons in the Wiley Series in Software Design Patterns. He has participated in a number of R. & D. projects and industrial projects. Uwe is (co-)author of open-source software systems, such as Extended Object Tcl (XOTcl), ActiWeb, Leela, Frag, and many others. He acts as a reviewer for journals and conferences. He has co-organized a number of workshops at conferences such as EuroPLoP, CHI, and OOPSLA. Uwe serves as conference chair for EuroPLoP 2005.

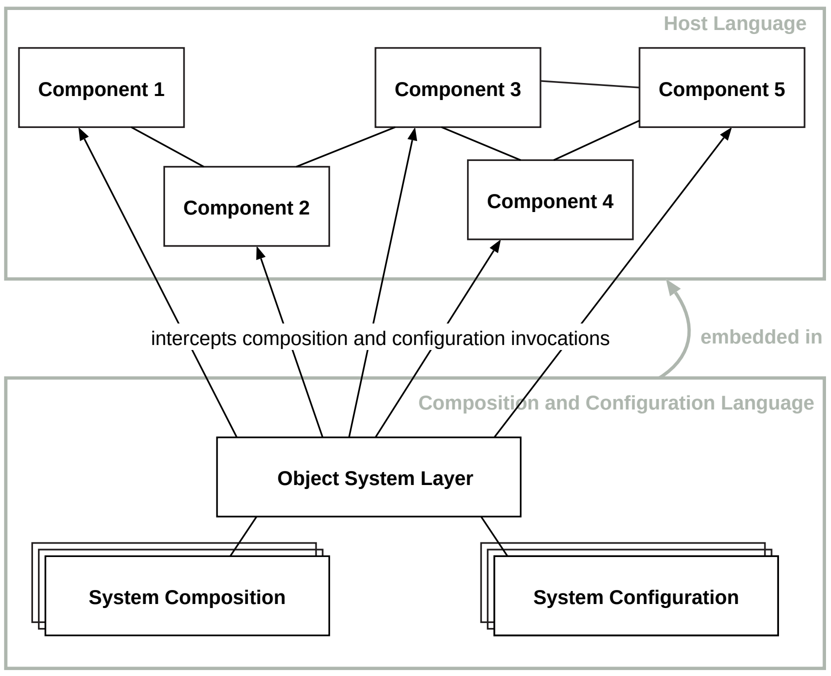

Figure 1. An object system layer connects the system’s composition and configuration components with the business logic components

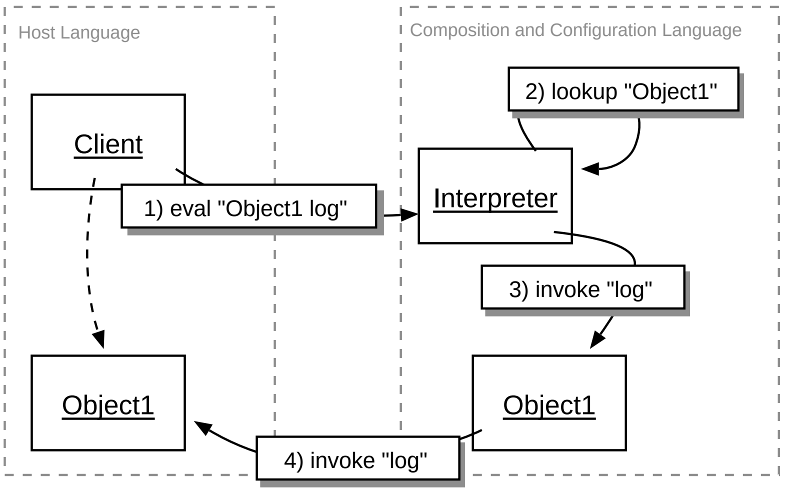

Figure 2. Split object example: An invocation from the host language cannot be dispatched in the composition and configuration language. Thus it is automatically forwarded to the host language object wrapped by the split object.

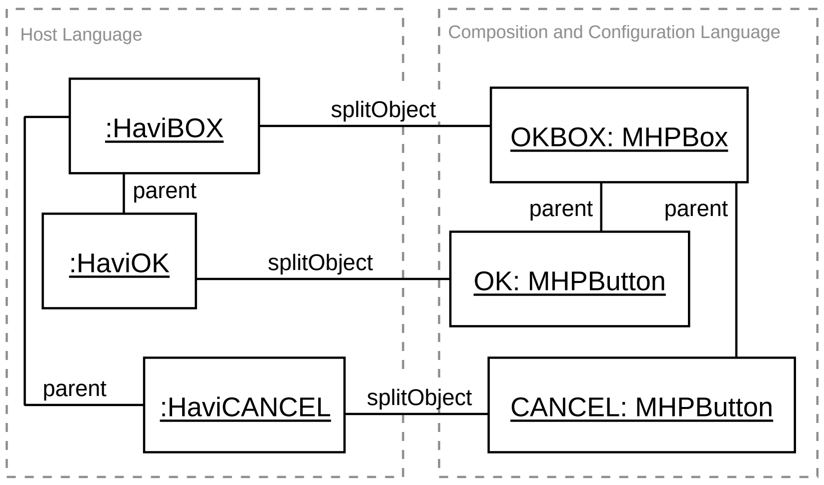

Figure 3. Object structure of the example

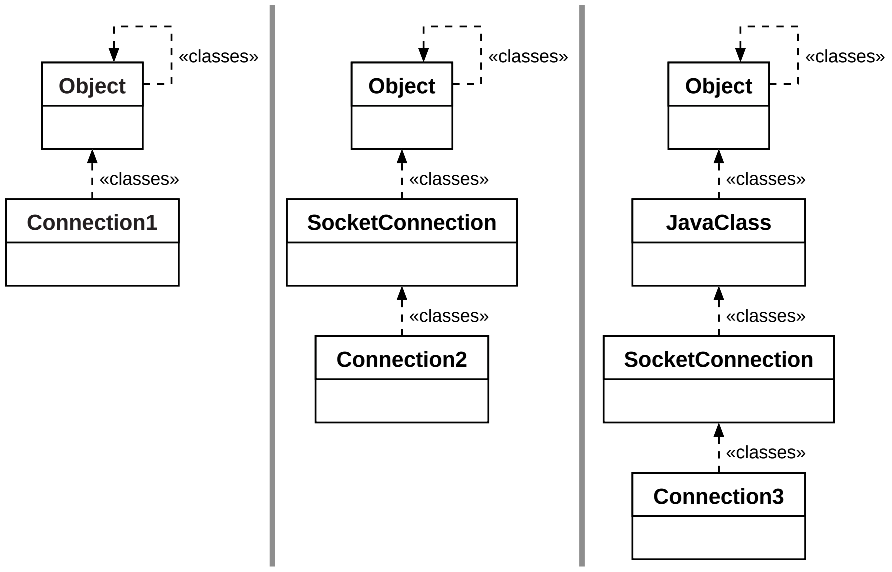

Figure 4. The three classes relationships in the examples

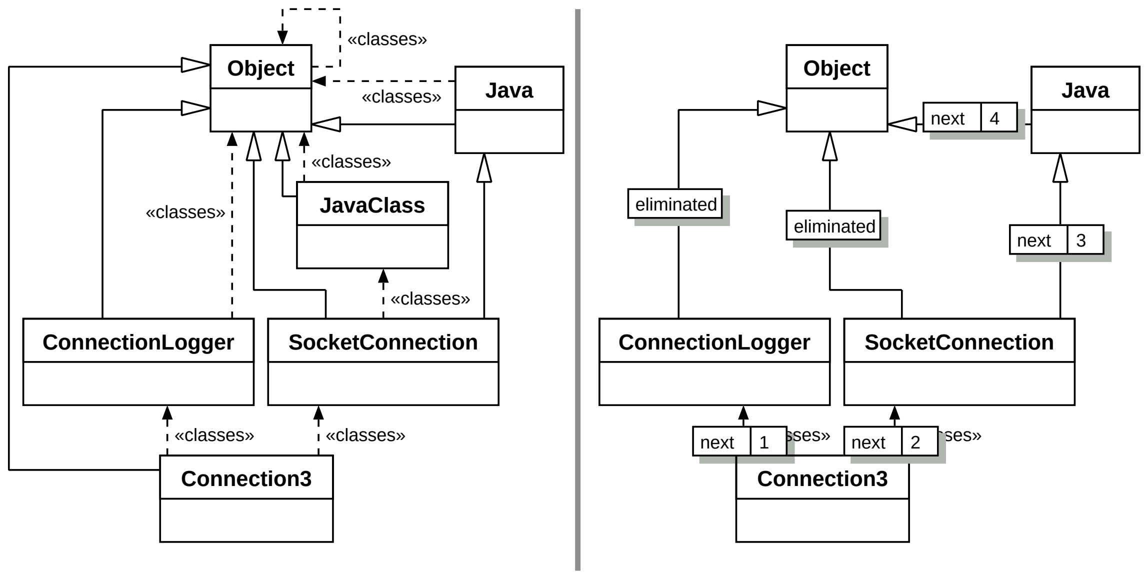

Figure 5. Left: full class and superclass graph; right: class and superclass graph with next path

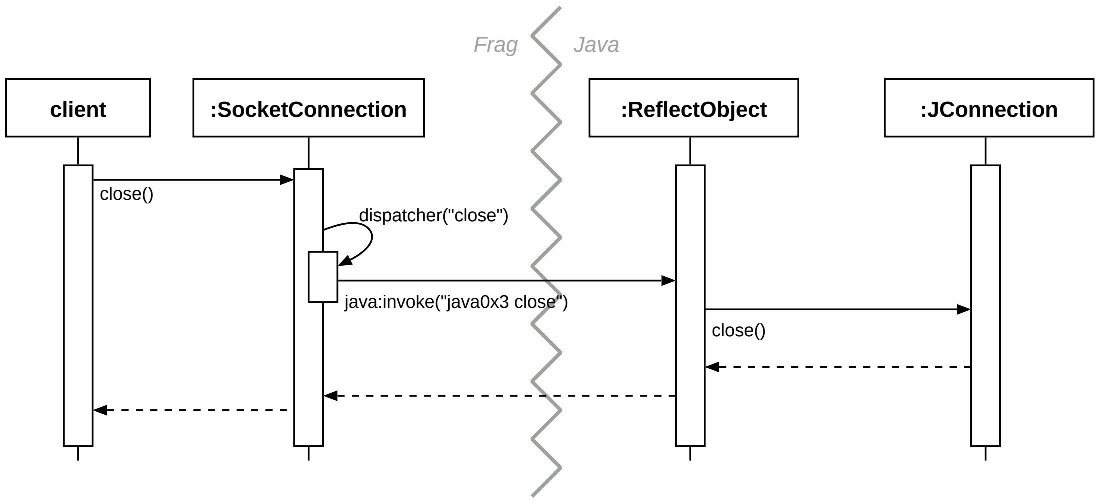

Figure 6. Dynamics of a dispatcher invocation to a Java proxy

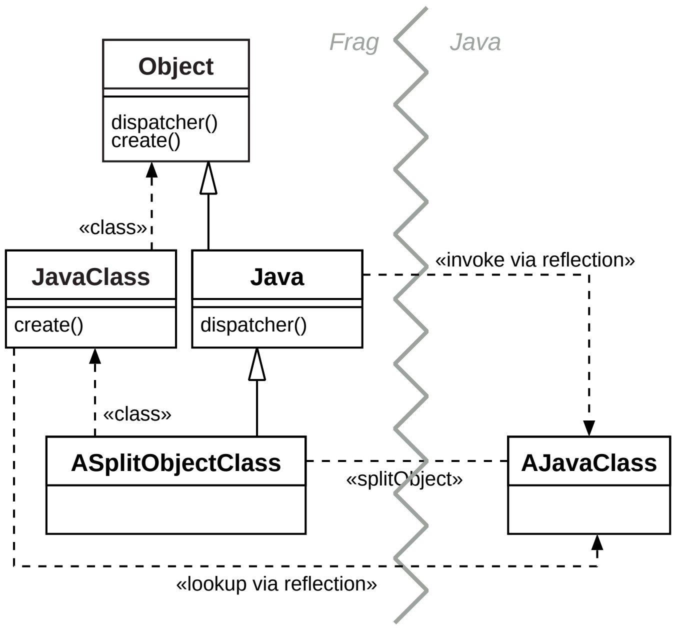

Figure 7. Split object structures example

Introspection Option	*Parameters*	*Description*
`args`	`method`	Parameter list of method
`body`	`method`	Body script of method
`children`		List of child objects
`classes`	`?pattern?`	Classes of an object
`default`	`method arg var`	Default value of a variable
`defaults`		Variable defaults of a class
`exists`	`var`	Returns `1` if `var` exists for an object, otherwise `0`
`instances`	`?pattern?`	Instances of a class
`nextpath`		Nextpath of an object
`parent`		Parent of an object, if any
`methods`	`?pattern?`	Methods of a class
`subclasses`	`?pattern?`	Subclasses of a class
`superclasses`	`?pattern?`	Superclasses of a class
`vars`	`?pattern?`	Variables of an object
`allInstances`	`?pattern?`	Recursively get all instances of a class
`allSubclasses`	`?pattern?`	Recursively get all subclasses of a class
`allSuperclasses`	`?pattern?`	Recursively get all superclasses of a class

Table 1

Introspection options

Callstack Information	*Parameters*	*Description*
`class`	`?level?`	Currently executing class (optionally: at given callstack level)
`method`	`?level?`	Currently executing method (optionally: at given callstack level)
`self`	`?level?`	Currently executing object (optionally: at given callstack level)
`level`		Current callstack level
`callingClass`		Class at calling level (equals `callstack class 1`)
`callingMethod`		Method at calling level (equals `callstack method 1`)
`callingObject`		Object at calling level (equals `callstack self 1`)

Table 2

Callstack information options