Using Split Objects for Maintenance and Reengineering Tasks

Uwe Zdun
New Media Lab, Department of Information Systems
Vienna University of Economics, Austria
zdun@acm.org

Abstract

Language integration is an important issue in the area of software maintenance and reengineering. We describe a novel solution in this area: automatically applied and composed split objects. Split objects provide a language integration that goes beyond simple wrappers by integrating object identity, state, methods, and class hierarchies of entities in two languages to one logical entity. The split object concept can be applied as an aspect-oriented solution, in which an aspect of a system is implemented in another language. After describing these concepts and two split object frameworks that we have implemented, we discuss how split objects can be applied for other maintenance and reengineering tasks than language integration. These application fields include software component testing, dynamic feature analysis, and variation and configuration management.

Keywords: Split Objects, Aspect-Oriented Programming (AOP), Wrapping, Language Integration, Testing, Dynamic Feature Analysis, Variation Management.

1. Introduction

Consider a situation in which a particular concern should be implemented in more than one programming language. This is a common situation in many maintenance or reengineering projects. A typical example is a legacy system, written in C or Cobol, that should be reengineered to the web. In the web application a Java-based application server should be used. Then the legacy language needs to be integrated with Java. Another typical reason for language integration is that one language is better suited for a particular task than the language in which the main application logic is (or should be) implemented. Ousterhout [23], for example, argues that dynamic and introspective scripting languages with strong string manipulation capabilities are good for component composition, writing test cases, and configuration tasks, whereas system languages, such as C, C++, or Java, are good for implementing the system's core, for instance, in form of reusable components. To implement each task in the best suited language, we need to integrate these languages somehow.

A typical solution for language integration in the area of software maintenance is to use wrappers. Wrappers are mechanisms for introducing new behavior to be executed before, after, in, and/or around an existing method or component. Wrapping is especially used as a technique for encapsulating legacy components [25]. Moreover, wrappers are used to extend object-oriented structures [3].

Even though wrappers are very useful in many situations, 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 language elements implementing the application logic. Examples of such semantics are internal 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 (i.e. the elements that are wrapped) is not explicit. In many cases, some of the system's data is required for the wrapper's client, but it is not accessible for the wrapper. To circumvent such problems, one can maintain such additional information on the wrapper in parallel to the application logic. But this is waste of performance and memory. In many cases, such solutions are implemented by hand (and thus hard to maintain).

In this paper we propose split objects as a wrapping approach that supports deeper language integration. A split object physically exist as an instance in both languages to be integrated, but logically it is treated like one, single instance. That is, invocations that cannot be dispatched for one of the two split object halfs can be delegated to the other half. This way, the methods and the state of one half can be accessed from the other one. 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, and methods can be wrapped. Optionally, (public) variables can be automatically traced; that is, a callback is executed to update a variable state on one split object half, when the same-named variable on the other half is changed. Depending on the language features of the two involved languages, these functionalities can either be implemented by extending the language's dispatch process, using reflection, or using generative programming techniques.

The very idea of split objects has been applied in a number of projects, for instance, in NS [28] and Open Mash [21] for integrating the object concepts of Object Tcl and C++. However, in these projects only a hand-built, hard-coded solution is applied (here the wrappers are registered in the C++ constructors of the wrappees). The main novel contributions of this work are:

- We have applied the split object concept for a number of maintenance tasks other than language integration, namely component testing, dynamic feature analysis, and variation and configuration management.
- We have integrated the split object concept with reflection [19] and generative programming techniques [7]. These techniques are used to automate the composition of a given system with split objects and the dispatch process between the wrapper and wrappee halfs of a split object. Compared to earlier split object solutions [28, 21], an important consequence of automating the wrapping process is that the wrappee half does not need to be changed (manually) for integrating with the wrapper. Generative aspect languages, such as AspectJ [16], can be used as a program generator as well (see Section 4).
- We have extended the split object concept to work as an aspect-oriented programming (AOP) [17] technique. Aspects avoid tangled solutions for cross-cutting design concerns. The typical situation in which split objects are applied is close to an aspect: a concern should not or cannot be implemented in one language, and thus it is implemented in a separate module (the aspect). Now the aspect needs to be composed with the system. This is a typical AOP situation, but unfortunately today's AOP composition frameworks such as AspectJ [16], JAC [24], Hyper/J [27], or JBoss AOP [4] are hard to apply in the named maintenance situations as a sole solution. Firstly, they do not deal with language integration but are solely working in one language (in the examples: Java). Secondly, aspect composition in these extensions is not dynamic and, as we argue later in this paper, this is crucial for some of the maintenance situations. Split objects can be used to implement dynamic, multi-language aspects or configuration of aspects on top of a given AOP implementation.

There are a couple of other approaches dealing with language interoperability. Interface description languages (IDLs) of middleware systems, such as CORBA's IDL or the Web Service Definition Language (WSDL), and the Inter-Language Unification (ILU) [31] are providing interoperability by invocations through well-specified, language-independent interfaces. .NET's common language runtime (CLR) can load and run code written in any programming language that is aware of the CLR. Interoperability is provided by a common language specification (CLS). Our approach is similar to these approaches in that it provides interoperability between different programming languages. However, our approach focuses on a different goal. It mainly provides a transparent and automatic indirection into a wrapper layer that fulfills additional tasks, such as the reengineering or maintenance tasks discussed in this paper.

This paper is structured as follows. At first, we explain the concept of split objects in Section 2. Next, in Section 3 we discuss two frameworks implementing this concept; one for integrating the Tcl extension Frag [32] with Java and one for integrating the Tcl extension XOTcl [20] with C++. Then, as an example of using split objects as an AOP concept, in Section 4 we discuss using AspectJ together with Frag to provide dynamic split object aspects for Java. In Section 5 we explain how to use these concepts and frameworks to implement the following software reengineering and maintenance tasks: component testing, dynamic feature analysis, and variation and configuration management. Finally, we conclude.

2. The Concept of Split Objects

Split objects are a concept to deal with the situation motivated in Section 1: because of some requirements a concern has to be implemented in more than one programming language. The concept of a split object is to make an object live across the language barrier by splitting it into two half objects, each of them living in one of the two languages. Logically, the two half objects share their identity and state. Typically the two languages run in the same process, even though this is not necessary.

2.1. Invoking Split Object Halfs

A split object can be accessed from within both languages. Both split object halfs are proxies [11] for the respective other half. That is, using a split object half the other half can be accessed.

Typically, there is a method `invoke` provided for a split object that allows one to sent messages to the respective other half. `invoke` receives the method name, the arguments, and the return type (if required) as parameters.

The `invoke` method can either be implemented by a superclass of the split object halfs or it can be introduced (for instance, using generative programming techniques or AOP) for each split object class. The implementation of the `invoke` method must be suited to the language features of the target language. There are the following options:

- *Dispatcher Extension*: If the target language contains a runtime dispatch mechanism (such as Tcl or Lisp for example), `invoke` can forward the invocation to the dispatcher and receive the result.
- *Reflection*: Some languages (such as Java or Tcl) contain a reflection API that allows one to look up a method by name and invoke it.
- *Generative Programming*: If the other options do not work (as in C or C++ for instance) or if these runtime techniques impose a problematic performance overhead, a program generator can be used to inject the invocations. For instance, to invoke C functions or C++ methods typically function pointers are registered that can be accessed using the `invoke` method. The disadvantage

of this technique is that only those methods can be invoked for which a function pointer binding was generated before. In other words, the runtime flexibility of this solution is more limited.

2.2. Automatic Type Conversion

For an invocation between two languages with differing type systems it is necessary to convert the types of parameter and return values. Usually a generic type is used for argument passing between the languages. For instance, we can use `Object` in Java, a void pointer in C/C++, and strings in scripting languages or distributed object systems.

For efficiency reasons, primitive types are converted using an automatic type converter. This is (partly) supported for many types by (some of) the programming languages. For instance, Java primitive types can be serialized to strings, and Tcl supports an automatic type converter for converting types to and from a string representation. Note that there are many ways to improve the efficiency of type conversion, such as lazy conversion or sharing of immutable objects.

Non-primitive types can be represented as split objects instead of converting them back and forth. That is, they exist in both languages and have a unique split object ID. Thus only the reference needs to be converted. This is very efficient, but it is not suitable for primitive types or instances not involved in split object interactions, because split object always impose a penalty in terms of performance and memory consumption.

Note that an automatic type converter is also provided for marshalling in distributed object frameworks, such as CORBA, RMI, Web Services, or .NET.

2.3. Automatic Forwarding

For most tasks it is useful that one of the two split object halfs is a wrapper for the other one. That is, the wrappee half contains some application logic, and the wrapper half contains an automatic forwarding mechanism to reach the wrappee half. Automatic forwarding is typically implemented using a number of elements:

- *Split Object Wrapper Superclass*: A superclass is assigned to all split object wrappers. It contains the implementation of the automatic forwarding mechanism. There is a field that stores the object identity of the wrappee. In non-OO languages the superclass concept can be simulated (see for instance [12]).

- *Split Object Factory*: A factory [11] allows one to create a split object from the runtime environment of the wrapper. The factory contains two factory methods. One makes an existing object in the wrappee's language a split object; that is, a wrapper is created. The other factory method creates a complete new split object, consisting of a wrapper and a wrappee.

- *Mixin Methods*: On the wrapper's superclass a method `next` (or sometimes called `proceed`) is defined that is able to forward a given invocation to the wrappee. This method is invoked within wrapper methods. At the point where this invocation takes place, the same-named method of the wrappee is invoked automatically. `next` returns the result of this method. The mixin method on the wrapper can perform behavior before or after the `next` invocation, it can omit the `next` invocation, and/or it can manipulate the result.

  In Figure 1 the dynamics of an invocation to a method `someInvocation` for the wrapper half of a split object is shown. The wrapper implements the method `someInvocation`. This method is a mixin method and contains an invocation of `next`. `next` forwards the invocation to `invoke`. `invoke` invokes the same-named method on the wrappee.

- *Variable Traces*: Optionally, variable traces can be supported. These allow the wrapper layer to bind a wrapper variable to a public variable of the wrappee. State changes are then automatically propagated.

- *Automatic Forwarding*: Some methods of the wrappee might not be altered in their behavior by the wrapper. It is tedious to hand-built a wrapper method for each such method on the wrapper. Instead the wrapper should automatically forward these methods. There are two variants to build an automatic forwarding mechanism:

  – *Default Behavior*: If the dispatcher of the wrapper's language can be extended (as in Tcl or Lisp, for instance), we can implement a default behavior for split object classes that is invoked every time a method invocation cannot be dispatched. The default behavior of a split object wrapper sends the same method invocation to wrappee split object half using the `invoke` method.

    In Figure 2 a method `someInvocation` is invoked for the wrapper half of a split object, but this method is not implemented for the wrapper. The dispatcher catches the runtime exception, and, as a default behavior, it tries to invoke the same-named method on the wrappee. This invocation either causes a runtime exception because the method is not found for the wrappee either, or it returns the invocation result. In the example in the figure, the result is returned to the client.

  – *Generated Wrapper Stubs*: In languages that do not allow to alter the dispatch process, we can simulate a default behavior by generating a wrapper stub for all methods that are defined for the wrappee, but not defined for the wrapper. These methods simply hand the invocation to the `invoke` method so that it gets executed on the wrappee.

2.4. Class Hierarchy Integration

Most often the split object layer mimics the user-defined class hierarchy of the wrappee half, because this way we can

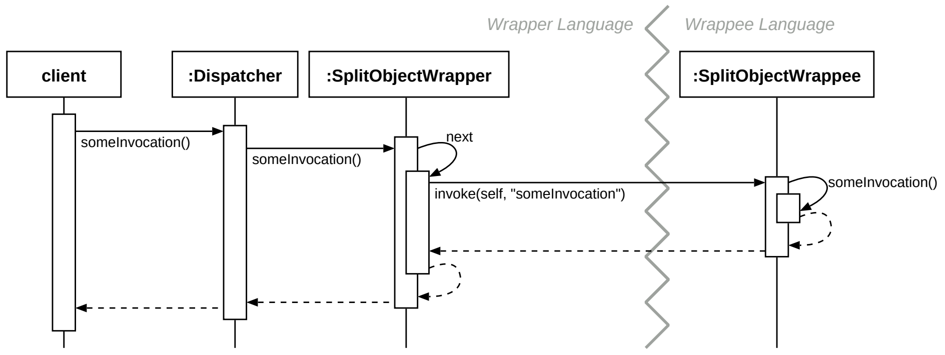

Figure 1. Method mixin dispatched for the wrapper half of a split object

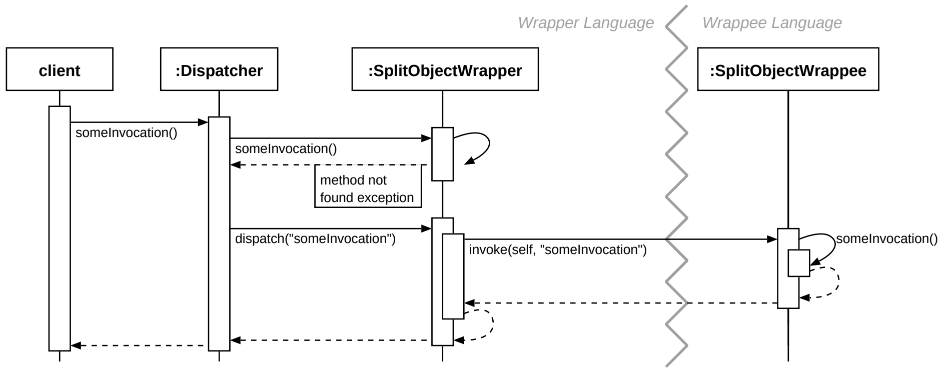

Figure 2. The default behavior of the wrapper half forwards unknown invocations to the wrappee half

use the same modeling means in the wrapper layer and in the application logic layer. Parallel class hierarchies are also very useful for structuring the method mixins, forwarder methods, and variable traces. A homomorphic structure is especially useful for generating wrapper stubs because we can introspect each class, and create the missing methods as wrapper stubs on its pendant in the wrapper layer. This way, name clashes can be avoided.

A structural homomorphism is not required for using the split object concept, but it helps to avoid the additional complexity of "new" structures in the wrapper layer. In general, only those parts of the host language class hierarchy are reflected in wrapper language that are needed for the wrapping task, and not more. For some tasks it even makes sense to build heterogeneous class hierarchies. Moreover, building parallel class hierarchies is not even always possible. For instance, when Java wraps C++ we cannot directly mimic multiple-inheritance relationships. Also, when non-object-oriented languages are used for wrapping, it is necessary to find a different solution.

An important part of class hierarchy integration is integrating the life-cycle concepts of the languages to avoid "unsafe" replication of objects or memory leaks. For instance, when integrating with Java we have to make sure that a reference to the Java instance is maintained as long as the wrapper language instance exists. When the object in the wrapper language is destroyed, all references to the Java object are deleted so that it can be garbage collected. In C++, in contrast, we have to invoke the respective destructor. Note that this one-to-one integration of the life-cycle of objects is only one possible alternative. Instead we might also implement split objects that are shared or instantiated on demand to optimize resource allocation.

3. Two Split Object Frameworks

We have explained the split object concepts on a fairly abstract level in the previous section because there are many different implementation variants for the different elements of a split object framework. Which implementation variants are best suited is highly depending on the features of the two languages that are integrated using split objects. In this section, we present two frameworks using object-oriented Tcl variants together with Java and C++ respectively, one implemented with reflection and one with a program generator. We use Tcl here, because we needed it in our projects; any other (object-oriented or procedural) languages can also be used for implementing the wrapper half of a split object framework.

3.1. Introducing Frag into Java using Java Reflection

Frag [32] is an object-oriented Tcl variant, completely implemented in Tcl. It is intended to be used as a configuration language for other languages. The Frag implementation runs in a Java Virtual Machine (on top of Jacl [8]), and it also works with the standard Tcl implementation implemented in C.

Within Frag, there is a Frag class `JavaClass` provided. This class is used for wrapping a Java class with a Frag object. The `create` method of this class creates a split object, consisting of a Java and a Frag half.

All split objects in Frag inherit from a class `Java`. The method `dispatcher` of the class `Java` is responsible for forwarding all invocations that cannot otherwise be dispatched to the Java half. `dispatcher` is automatically invoked as a default behavior, when a method cannot be dispatched in Frag.

Internally, primitive Java types are automatically converted to and from strings. Non-primitive types have to be used as split objects in order to be used from Frag.

Consider we want to put a Java `Integer` objects into a Java `Hashtable`. First, we have to create wrapper classes for the respective Java classes:

```
JavaClass create Integer
JavaClass create Hashtable \
  -set javaClass java.util.Hashtable
```

The Java classes can be used to create split objects from within Frag. For instance, we can create a Java Hashtable and an Integer instance:

```
Hashtable create ht
Integer create i1 1
```

The object names (`ht` and `i1`) identify split objects. These can be used to invoke methods via Java reflection. Actually in the above example we have already invoked the Java constructors this way. Note that the Java `Integer` constructor requires an initial value as argument: internally the string `1` is recognized as an integer and is automatically converted to the Java type `int`, required by the constructor.

As an example for an ordinary invocation, we can `put` the Integer into the hashtable using the string ID `first`:

```
ht put first i1
```

Figure 3 shows the class relationships for the `Hashtable`.

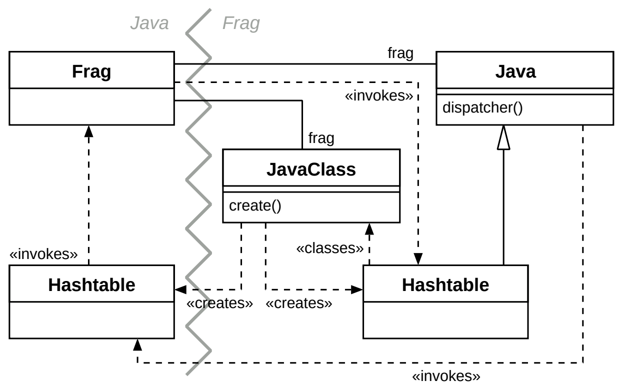

Figure 3. Example Class Relationships

3.2. Introducing XOTcl into C++ using SWIG

XOTcl [20] is an object-oriented variant of the language Tcl, implemented in C. It is typically used for gluing (and objectifying) C components. In principle, it can also be used for gluing components written in C++, but without further support C++ would have to be integrated in the same way as C: by wrapping C++ methods with (hand-written) wrappers that are using function pointers. But this way we would loose much information that can be well modeled in XOTcl such as C++ method, class, and inheritance structures. Thus a better idea is to “really” integrate the two object-oriented languages using the split objects concept.

We cannot use a solution similar to the Java solution, explained in the previous section, because C++ does not allow for reflective lookup and invocation of methods. Instead, the XOTcl wrapper classes and invocations can be automatically generated. In our implementation we used the wrapper generator SWIG [26] to create all wrapper classes. Our SWIG-XOTcl implementation automatically creates XOTcl classes and C++ wrapping code from a given SWIG interface file. Thus we only have to document the C++ interface as a SWIG interface file, and all necessary wrapper code is generated automatically. For instance, consider we want to use XOTcl scripts to access a simple C++ stack class. First, a SWIG interface file for this C++ class is needed. Note that this file is almost identical to a C++ header file:

```
%module stack
%{
#include "stack.h"
%}
class Stack {
  public:
    Stack();
    ~Stack();
    void push(void* element);
    void* peek();
    void* pop();
};
```

Next the wrapper generator is run. It produces wrapper functions for accessing the C++ public methods declared in the interface file, including the constructor and destructor. It also creates an XOTcl class `C++`, responsible for connecting an XOTcl object to a C++ instance. That means, it stores an ID for the C++ instance, and creates or destroys the C++ instance, when the XOTcl instance is created or destroyed.

For each class name in the SWIG header file a same-named XOTcl class is generated that inherits from the `C++` class. For each C++ method, the XOTcl wrapper class has a same-named method that invokes the C++ method using a function pointer. For each variable a Tcl variable trace is produced. This trace sets the C++ variable whenever the same-named variable is accessed from Tcl.

4. Introducing AOP into Frag/Java Split Objects using AspectJ

As mentioned before, the situations where split objects can be applied are close to AOP, because the split object wrapper untangles a concern from an implementation. If used as an ordinary wrapper implementation, the client invokes the wrapper. The wrapper forwards the invocation to the wrappee (as depicted in Figures 1 and 2). This alone is not an AOP solution: in the split object solution explained so far, the client needs to know that the split object is applied, because it has to invoke the wrapper. What is missing is a means to recompose the separated concern with the application logic in a non-invasive way.

As a simple example, where such a non-invasive composition is important, consider again the Frag/Java split object solution. In some cases, we might want to use Frag to make additions to an existing Java program. Consider a `hashtable` is defined as a split object, and the following invocation is performed in the host language Java:

```
hashtable.get("first");
```

Even though the `hashtable` is defined as a split object, this invocation bypasses the wrapper half and reaches the Java `hashtable` object directly. To avoid bypassing we need to modify the client or wrappee implementation. Instead of dispatching the invocation to the wrappee split object half directly, the invocation is performed for the wrapper half first. What would be needed is an invocation sent through the wrapper layer:

```
frag.eval("hashtable get first");
```

It is tedious to insert such invocation into the source code. A transparent and automatic invocation of the wrapper half, as depicted in Figure 4, is required. The wrappee half of the split object is called by a client. The invocation is not dispatched directly, but another operation `dispatch` is called that indirects the invocation into the wrapper layer. This forwarding step should not be hand-built, but “woven” into the application. Thus forwarding is not visible to the client (i.e. the change is “non-invasive”). In the wrapper layer a dispatcher forwards the invocation to the respective split object half. This object performs the invocation. During the invocation it invokes the primitive `next` that lets the dispatcher perform the original invocation on the wrappee. The wrapper receives the result, and it can handle this result in arbitrary ways. For instance, it can simply return it to the client.

We have developed an AspectJ aspect that weaves Frag split objects into a given Java program. The aspect contains one advice that is invoked before the constructors of user-defined classes. It calls `makeSplitObject` that creates a split object half in Frag. All other methods of the user-defined classes are intercepted by an around advice. The method `invokeSplitObject` sends the invocation to the split object half first, and if `next` is invoked, it is sent to the Java implementation as well:

```
abstract aspect FragSplitObject {
  static Frag frag;
  ...
  static void makeSplitObject(JoinPoint jp,
    Object o) {...}
  static Object invokeSplitObject(JoinPoint jp,
    Object o) {...}
  ...
  before(Object obj): theConstructors(obj) {
      makeSplitObject(thisJoinPoint, obj);
  }
  Object around(Object obj) : theMethods(obj) {
      return invokeSplitObject(thisJoinPoint,
                               obj);
  }
}
```

The aspect is defined as an abstract aspect. In concrete aspects the pointcut `splitObjectClasses` is refined. That is, the classes to which the aspect is applied can be defined by the user. For instance, we can apply the split objects to the classes `A`, `B`, and `C`:

```
public aspect FragSplitObjectABC
  extends FragSplitObject {
  pointcut splitObjectClasses(Object obj):
    this(obj) &&
    (within(A) || within(B) || within(C));
  ...
}
```

Note that we have used AspectJ as a static AOP framework for Java to aid the implementation of split object aspects. The use of AspectJ for this purpose is not mandatory, we can use most other aspect frameworks for weaving-in the forwarding step. For instance, we can use another program generator such as or Compost [1] to weave the split object into the wrappee code. If the source code is not available, we can use a byte code manipulator such as Javassist [6] to weave the aspect into the byte code. In languages that support dynamic message interceptors (such as XOTcl [20], Frag [32], and many distributed object systems), we can use the interceptors to indirect messages to the split object. Once applied, the split object aspect solution goes beyond the original (static) AOP solution in two ways that are important for reengineering or maintenance tasks explained in Section 5:

- The split object aspect introduces the language properties of the wrapper language into the host language and its AOP framework. When using Frag as the wrapper language for Java and AspectJ, for instance, we introduce Frag’s language dynamics into the statically woven AspectJ aspect, written in Java. That is, we can dynamically configure and compose the aspects at runtime.
- The code written in the wrapper language can be reused across different host languages. For example, we can reuse Frag code across different host languages supported by Frag (Java, C, C++, and Tcl).

The default behavior of the split object aspect, explained above, is that all invocations are sent through the split object layer, but the invocations are not altered. By re-defining the classes in the dynamic Frag language, one can dynamically configure and compose the aspect. For example, if we want to redirect all invocations of a method `open` on class `A` to an instance of a Java class `Logger`, we have to create a split object binding for the `Logger` class and introduce the indirection method for class `A` dynamically. This mixin method first logs the invocation and then invokes the original method behavior using `next`:

```
String script =
  "A method open {logger logMethod A open; next}";
frag.eval(script);
```

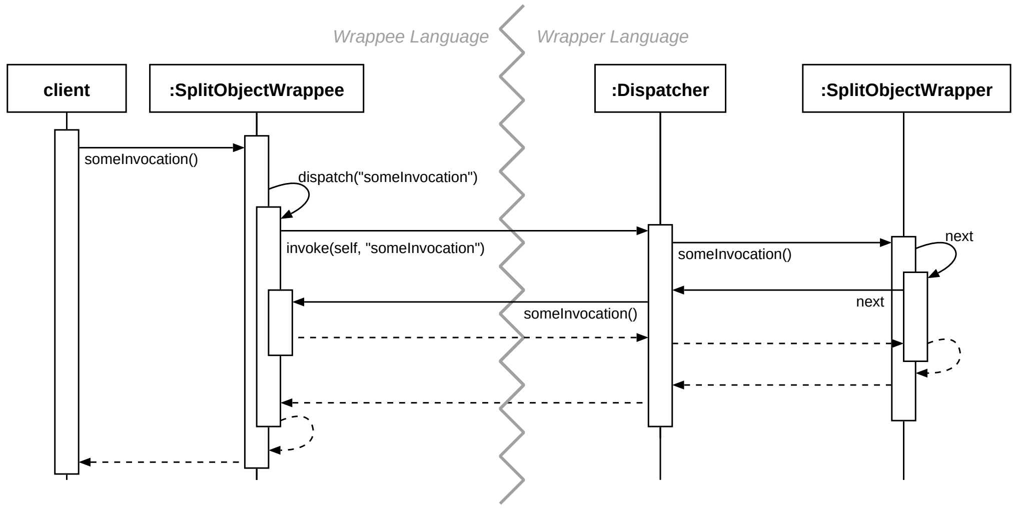

Figure 4. A split object aspect is woven into the wrappee code

5. Using Split Objects for Reengineering and Maintenance Tasks

Now that we have introduced the concept of split objects, two implementation variants, and split object aspects, we want to discuss some possible uses of the concept and the frameworks in the field of software reengineering and maintenance.

5.1. Component Testing

An important software maintenance task is software testing. Some approaches aim at easing the process of writing tests. For instance, JUnit [10] is a framework that eases writing test cases in Java and assembling them into test suites. A drawback of this approach is that it requires the developer to build the test cases apart from the components to be tested. Software tests, however, are conceptually closely coupled to the components to be tested. On the other hand, tightly coupling test cases with implementation details is problematic as well, because as soon as the implementation details change, the test cases have to be adjusted as well. If the number of test cases grows, test case maintenance can become a serious concern. JUnit only offers a programmatic compilation of test cases in Java requiring recompilation for reconfiguration. Many projects prefer scripting languages for test case management and specification because of the rapid changeability and the string manipulation capabilities of these languages.

Different approaches aim at modeling a coupling between components and component tests. For instance, Orso et al. [22] derive assertion-based self-checks of software components using component metadata. Some approaches propose built-in tests for components [29, 2] that are part of the class specification. Jézéquel et al. apply a design-by-contract approach to implement self-testable software components [15]. In particular, they embed “test-contracts” in source code comments and apply a preprocessor to extract the test information before compilation. These approaches, in turn, have the drawback that they bloat the software components’ sources with metadata or test code.

The problem that tests should be separated from the component to be tested, but yet a close coupling is required, lets an AOP solution come to mind. Using AspectJ [16], for instance, one can “introduce” test methods to existing classes. This way tests are separated in the aspect, but still share the object identity with the instance to be tested.

Unfortunately, the current generative AOP frameworks, such as AspectJ [16] or Hyper/J [27], have a number of problems regarding component testing. For instance, the pointcut languages of most AOP frameworks do not allow an aspect to be (easily) composed with a particular instance. Also it might be hard to specify context constraints using pointcuts. Dynamic aspect composition at runtime is not supported, and thus re-compilation is required to change test cases.

Split objects provide a solution balancing the forces discussed above. We have applied these concepts for testing C and C++ (and XOTcl) components. In our concept, we use component metadata in the scripting language XOTcl to specify the test information. This metadata describes the split object wrapper hierarchy, automatically generated by SWIG. A test framework is provided that introspects these classes for test metadata. The whole test suite can be run automatically.

Consider for instance a simple C++ `Counter` class. The SWIG wrapper generator creates a split object class `Counter` in XOTcl. In the test script we can derive an instance from this class, and test it. The methods `count` and `counter` are only defined in C++. SWIG generates wrapper methods so that the XOTcl split object half forwards invocations sent to these methods into C++.

The test framework executes the `testScript`, and automatically compares a user-defined `resultScript` with the result of the last invocation in the test script (which returns the counter value in the example below):

```
@Test counting::countTo5 {
  testScript {
    Counter create c1
    for {set i 0} {$i < 5} {incr i} {
      c1 count
    }
    c1 counter
  }
  resultScript {5}
}
```

The split object solution for component testing has a number of benefits that are not offered in their entirety by any of the other approaches discussed before. The split object shares the identity with test object; thus there is a close coupling between test and tested object. The test wrapper is annotated with component metadata for tests, instead of hard-coding test cases. The test code is completely untangled from the code of the production system. Test cases can be specified in an invasive manner (e.g. for a specific instance) or non-invasive manner (e.g. using introspective queries like "apply the test for all instances whose class name begins with XYZ"). The split object code can be completely generated by SWIG automatically. Test cases are written in the flexible scripting language with powerful string processing capabilities. Thus it is very easy to write test cases rapidly. The test framework, written in XOTcl, can be reused for many languages (Tcl, XOTcl, C, and C++).

There are some drawbacks of the XOTcl split object solution for testing C++ as well. Some additional work is required for specifying the SWIG interface files of the C++ components. The approach imposes some overheads in terms of memory and performance usage. For typical test cases this should not pose a problem. But in cases where the test suite needs to be part of the operational system this might be problematic. Developers have to acquire a basic Tcl knowledge for test case specification.

5.2. Improving Dynamic Feature Analysis

For reengineering a large-scale system it is crucial to gain an initial understanding of the system and its architecture. Even though there are many tools and approaches to completely reverse engineer an existing system [18], in many cases these might produce very complex results or require a significant amount of work. Static feature analysis such as semi-automatic feature location based on dependence graphs [5] also poses some problems, especially regarding complexity in large-scale systems that are not well documented.

Dynamic feature analysis techniques [30, 9] can provide important starting points for static program analysis. Gschwind and Oberleitner [14] identify some shortcomings in current dynamic feature analysis techniques. For each new analysis a program run is required, sometimes even requiring a complete re-compilation. Traces cannot be tuned to particular objects or method invocations. Parameter values passed to objects are not available during analysis.

To overcome these shortcomings Gschwind and Oberleitner propose ARE, a reengineering tool based on AspectJ. The general idea of the tool is to compose an aspect with all classes to be traced, and use the dynamic joinpoint model of AspectJ for fine-tuning these trace aspects at runtime. This solution avoids the shortcomings named above, but yet it has some shortcomings as well. A main drawback is that AspectJ's aspect instrumentation is not really object specific or method invocation specific. The `thisJoinpoint` invocation context of AspectJ is used to obtain method signatures and arguments at runtime. The aspect woven into the application remains unchanged for the whole program execution. As a consequence any manipulation of the control flow (such as adding a new AspectJ advice) or the structures (such as adding an AspectJ introduction) requires recompilation of aspects and the system.

Note that exactly this problem is addressed by injecting Frag split objects into Java using AspectJ, as explained in Section 4. The system is instrumented so that every invocation is firstly sent through the split object layer. After instrumentation, the split objects are just wrappers that are invoked instead of the original receives, and forward all invocations to these original receivers (they are simply invoking `next`). Thus the system is working in the same way as before, but every invocation passes the split object layer.

In the split object layer we are able to trace and manipulate the split objects. Using the introspection options of the scripting language, we can also find out about the static structure of the entities involved in a control flow.

The object half in the scripting language can be changed at runtime, once it is composed with the system. Split objects allow for dynamic feature analysis in the style of ARE: traces of control flows with all parameters and results can be analyzed and specific traces can be filtered out using method mixins. Consider for instance a method `x` of a Java class `B` is invoked, but only if an instance of another class `A` has invoked it, the method should be logged. This fine-tuning can be reached by introspecting the calling object of the invocation and the type of this object.

```
B method x {} {
  if {[[callstack callingObject] info isType "::A"]}
    Logger writeTrace [list Called [self] \
      [callstack method]]
  }
  next
}
```

Note that we have used control flow data (the callstack information `callingObject`) and structure data (the `isType` introspection option) to fine-tune feature logging in this wrapper method.

There are some interesting features for manipulating a system. These can be very helpful when restructuring a system. A split object can be used to dynamically introduce new methods to a class. Using method mixins we can inject method calls. That is, the control flow can be changed from the within the split object layer. For instance, we can redirect an invocation to a different instance than the one originally called (without needing to recompile the system to add such manipulations).

The split object solution is not well applicable for all dynamic analysis tasks. Any analysis that depends on performance measurements suffers from the overheads of the split object layer and are thus not accurate (the same applies for other AOP solutions). AOP techniques typically have less instrumentation points than pure parse tree approaches, because the pointcut and joinpoint models abstract from these details. Compost [1] operates directly on the parse tree and thus might be an alternative when a more fine-grained instrumentation is needed. In general, our approach does not rely on AspectJ for instrumentation: any framework can be used that can intercept the construction of an entity (to make it a split object) as well as all invocations (to indirect them into the split object layer). That is, our approach can also be applied to other languages than Java or to the Java Byte Code (e.g. using Javassist [6]).

5.3. Variation and Configuration Management

Many monolithic systems have problems regarding introduction of new variants, flexibility, and configuration management. This problem occurs for reengineering of legacy systems, as well as the maintenance of newly developed system. We have experienced this problem for instance during reengineering a C-based document archive system [13] and for variation management of software for Java-based interactive television set-top boxes. In both cases, hard-coding the variations or configuration options was not enough, because rapid customization without re-compilation (even by non-programmers in some cases) was required.

Many projects use metadata in a separate file, as for instance an XML configuration file, as a solution. For instance, JBoss AOP uses XML configuration files to configure aspect composition [4]. This solution is good for handling simple, structured configuration options or variations in a declarative manner. It is hard, however, to deal with configuration options or variations that require behavioral specifications and/or programming constructs, such as conditions, loops, blocks, substitutions, or expressions. JAC [24] allows an aspect to define its own configuration options. Even though this works for simple behavioral configurations, a real programming language is substantially more expressive than these simple domain-specific languages.

In such cases, we propose to use split objects and a dynamic configuration language instead. For instance, Frag is designed for configuring Java using scripts. Consider, for instance, interactive games that should run on the digital television set-top box. A programming language is needed, but yet programming in-game scenes in Java is tedious. Game level and scene designers usually are not 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.

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]
}
```

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 such parameters, there would be a considerable overhead in terms of development times. Instead it makes sense to dynamically manipulate and exchange the script 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 most current AOP languages require re-compilation. The split object solution has the disadvantages that the embedded interpreted language is slower than a compiled solution.

The problem of having to add variation management (like versioning for instance) or configuration options, often occurs during legacy reengineering as well. Many AOP languages are not applicable in this context because for many legacy system languages there are no stable AOP language extensions (yet). Here, the split object solution has the advantage that it is significantly easier to develop an in-house solution of (the wrappee part of) a split object framework than an AOP language extension for the legacy system's language.

5.4. Integration with Non-OO Languages

So far we have discussed the approach from an object-oriented perspective because split objects are an object-oriented concept. Yet there is in no limitation regarding the host language, and object-orientation can be simulated in procedural languages (see [12]).

For instance, we have integrated C programs into object-oriented applications using split objects. Then a number of C functions has to be grouped to form an object-oriented wrapper as the wrappee half of a split object. In [13] we present the details of a solution for a C-based document archive system.

6. Conclusion

In this paper we have described split objects as a practical concept for language integration and extended the basic concept in various ways. In particular, we have developed a concept to automate the application and composition of split objects (either using reflection or generative techniques). As a proof of concept we have presented two split object frameworks for C++/XOTcl and Java/Frag. We have also described how to use split objects as dynamically configurable and composable aspects (a feature not offered by today's aspect composition frameworks solely). Note that the general concept is applicable for any programming language; it is not limited to using a Tcl variant as the wrapper language. We have used Tcl variants because they have provided the required dynamic and introspective language features in our application areas: component testing, dynamic feature analysis, variation and configuration management. There are a number of other potential application areas, such as calculating dynamic metrics. For these (quite diverse) reengineering and maintenance tasks we have shown that split objects can be applied to avoid some drawbacks of existing solutions. As outlined before, we have used the concepts with success in a number of practical research and industry projects.

The approach should only be applied if the flexibility of the split object layer is needed. In other cases, there is a performance and memory overhead because every invocation needs to be sent through split object layer. This overhead is rather hard to quantify in general because different languages and integration solutions have quite different properties. In most cases, the performance overhead equals an additional invocation in the (often slower) wrapper language. Another drawback of split objects is the higher complexity of the software structures and that developers have to learn two languages. Note that these issues are only relevant in comparison with a pure host language design. In appropriate application cases, where the split object flexibility is needed by the application task, however, pure host language solutions are often complex and slow as well. Thus a proven, reusable split object solution is typically easier to understand and maintain than an equally powerful solution in the host language that is implemented for a particular project only.

References

[1] U. Aßmann and A. Ludwig. Introducing Connections into Classes with Static Metaprogramming. In P. Ciancarini and A. Wolf, editors, *3rd Int. Conf. on Coordination*, number 1594. Springer, Apr. 1999.

[2] C. Atkinson and H. Gross. Built-in contract testing in model-driven, component-based development. In *Proc. of ICSR-7 Workshop on Component-Based Development Processes*, April 2002.

[3] J. Brant, R. E. Johnson, D. Roberts, and B. Foote. Evolution, architecture, and metamorphosis. In *Proc. of 12th European Conference on Object-Oriented Programming (ECOOP'98)*, Brussels, Belgium, July 1998.

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

[5] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension*, June 2000.

[6] S. Chiba. Javassist. http://www.csg.is.titech.ac.jp/∼chiba/javassist/, 2003.

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

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

[9] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the International Conference on Software Maintenance*, November 2001.

[10] E. Gamma and K. Beck. JUnit. http://www.junit.org/, 2003.

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

[12] M. Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceedings of EuroPlop 2000*, pages 397–410, Irsee, Germany, July 2000.

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

[14] T. Gschwind and J. Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Proccedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR2003)*, Benevento, Italy, March 2003.

[15] J. Jézéquel, D. Deveaux, and Y. L. Traon. Reliable Objects: Lightweight Testing for OO Languages. *IEEE Software*, 18(4), July/August 2001.

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

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

[18] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, University of Stuttgart, 2000.

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

[20] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, pages 163–174, Austin, Texas, USA, February 2000.

[21] Open Mash Consortium. The open mash consortium. http://www.openmash.org, 2000.

[22] A. Orso, M. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. In *2nd Int. Workshop on Engineering Distributed Objects (EDO 2000)*, Davis, USA, Nov 2000.

[23] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31, March 1998.

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

[25] H. M. Sneed. Encapsulation of legacy software: A technique for reusing legacy software components. *Annals of Software Engineering*, 9, 2000.

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

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

[28] UCB Multicast Network Research Group. Network simulator - ns (version 2). http://www.isi.edu/nsnam/ns/, 2000.

[29] Y. Wang, G. King, and H. Wickburg. A method for built-in tests in component-based software maintenance. In *IEEE International Conference on Software Maintenance and Reengineering (CSMR'99)*, pages 186–189, March 1999.

[30] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7:49–62, 1995.

[31] Xerox. Inter-Language Unification. http://www2.parc.com/istl/projects/ILU/, 1999.

[32] U. Zdun. Frag. http://frag.sourceforge.net/, 2003.