Some Patterns of Component and Language Integration

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

Integration is an important concern in many software systems. In this paper, we present a number of patterns that are used to improve the integration of a system with components or code that is written in different languages than the system itself. Component integration is necessary when (foreign) components should be used within a system. The challenge of integrating components into a system is that heterogeneous kinds of components exist, perhaps without distinct interfaces or other component boundaries. The task of the component integration code is to provide suitable, stable invocation interfaces and to compose the components with the system. Sometimes, however, invocation and composition of components is not enough, but a deeper language integration is required. Examples of what might be need are automatic type conversions between languages, preserving the destruction order of the other language, automatic forwarding of invocations into the other language, and runtime integration of foreign language components. The patterns, presented in this paper, are successful solutions in these areas. Each of these patterns plays an important role in different integration architectures. We will give examples from the areas of distributed systems, reengineering projects, scripting languages, and aspect-oriented programming systems. There are many others fields where the patterns are used as well.

Introduction

Software Integration

Integration of software systems is needed in many situations. Just consider two enterprise applications that need to be integrated after a fusion of the companies took place. Or, consider a reengineering project in which a terminal-based legacy system should be re-engineered to the Web – and thus must be integrated with a modern application server. The capability of the system's architecture to integrate foreign components and to be integrated with other systems is thus considered as an important quality attribute [BCK98]. There are many integration concerns to be considered when integrating software architectures, including:

- integration of data, data models, and queries,
- integration of various software components,
- integration of various programming languages,
- integration of distributed systems,

- integration of system with legacy systems, and
- integration of different software paradigms such as the object-oriented and procedural paradigm.

Pattern Language Outline

In this paper, we present patterns that describe successful solutions in some of these areas. These patterns are not a complete pattern language for the broad field of software architecture integration – rather, they are an excerpt working in the area of component, language, and paradigm integration mainly. There are also many patterns from other pattern collections that interact closely with the patterns presented in this paper.

The patterns presented in this paper are applied where simple, ad hoc integration solutions fail. Thus the main audience of the patterns are experienced software architects and developers.

Patterns for Component and Language Integration

In this section, we provide short thumbnails for the patterns and important external patterns, as well as a pattern map as an overview.

Pattern Thumbnails

The following patterns are presented in this paper:

- A COMMAND LANGUAGE is a language that is implemented within another language (called "host language"). It uses COMMANDS [GHJV94] implemented in that host language as implementations of its language elements. The COMMAND LANGUAGE lets the COMMANDS be assembled in scripts, and it provides an INTERPRETER [GHJV94] or another interpretation mechanism such as an on-the-fly compiler for these scripts.
- A COMPONENT WRAPPER is an object implemented as part of a component that represents another component. The wrapped component is imported into the wrapping component, and the COMPONENT WRAPPER handles all details of the import. For the integration it utilizes other patterns, such as WRAPPER FACADE [SSRB00] or PROXY [GHJV94]. The COMPONENT WRAPPER is an accessible white-box representing the component within another component's scope, where adaptations such as interface adaptations or version changes can take place.
- An OBJECT SYSTEM LAYER [GNZ00] implements an object systems as a LAYER [BMR$^+$96] 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. A part of an OBJECT SYSTEM LAYER is a MESSAGE REDIRECTOR [GNZ01] that forms a FACADE [GHJV94] to the object system; all invocations into the OBJECT SYSTEM LAYER from the rest of the system are sent to its MESSAGE REDIRECTOR and dispatched here to the objects in the object system.
- An AUTOMATIC TYPE CONVERTER converts types at runtime from one type to another. There are two main variants of the pattern: One-to-one converters between all type bindings and converters utilizing a canonical format to/from which all supported types can be converted. AUTOMATIC TYPE CONVERTER can, for instance, be used in a COMMAND

LANGUAGE to convert types to/from the host language, in SPLIT OBJECTS to convert between the SPLIT OBJECT halfs, and in COMPONENT WRAPPERS to convert invocations to the types of the wrapped component and vice versa.

- A SPLIT OBJECT is an object that is split across two languages. The SPLIT OBJECT is treated logically like a single instance, but it is implemented by two physical entities, one in each language. That is, both halfs can forward invocations to the other half, access the inner state of the other half, and usually – to a certain extent – the user-defined class hierarchies of each half is mimicked by the respective other half.

External Patterns There are a number of external patterns, documented elsewhere, that play an important role for the patterns described in this paper. We want to explain some of these patterns briefly:

- A COMMAND [GHJV94] encapsulates an invocation to an object and provides a generic (abstract) invocation interface. COMMANDS alone only allow for adaptation by changing the association link to a COMMAND. In the COMMAND LANGUAGE pattern the COMMAND abstraction is extended with a binding of COMMANDS to language elements and an interpretation mechanism for that language.
- The important interpretation step in a COMMAND LANGUAGE can be implemented using another pattern from [GHJV94], the INTERPRETER pattern. In general an INTERPRETER defines a representation for a grammar along with an interpretation mechanism to interpret the language.
- The pattern INTERFACE DESCRIPTION [VKZ04] describes how to specify interfaces of an object so that external clients of the object can reliably access the object. The pattern is originally described in the context of distributed systems: In this context an INTERFACE DESCRIPTION specifies the interface of a remote object for remote clients, but can also be used in other scenarios. Generating type conversion code from an INTERFACE DESCRIPTION is the most important alternative solution to using an AUTOMATIC TYPE CONVERTER. AUTOMATIC TYPE CONVERTERS can also be applied according to an INTERFACE DESCRIPTION.
- A MESSAGE REDIRECTOR [GNZ01] receives symbolic invocations – for instance provided in form of strings – and redirects these invocations to objects implementing the invocation behavior. This way, for instance, a dynamic object dispatch mechanism can be implemented. A MESSAGE REDIRECTOR can also be used to integrate some non-object-oriented invocation interface with an object system. The MESSAGE REDIRECTOR pattern is an inherent part of an OBJECT SYSTEM LAYER: It is used to dispatch the invocations of OBJECT SYSTEM LAYER to its objects.

Figure 1 shows an overview of the patterns described in this paper and the relationships explained above.

Command Language

Context Some code needs to executed by other code and needs to be exchangeable at runtime. In static languages you cannot simply exchange code. In such situations, COMMANDS provide

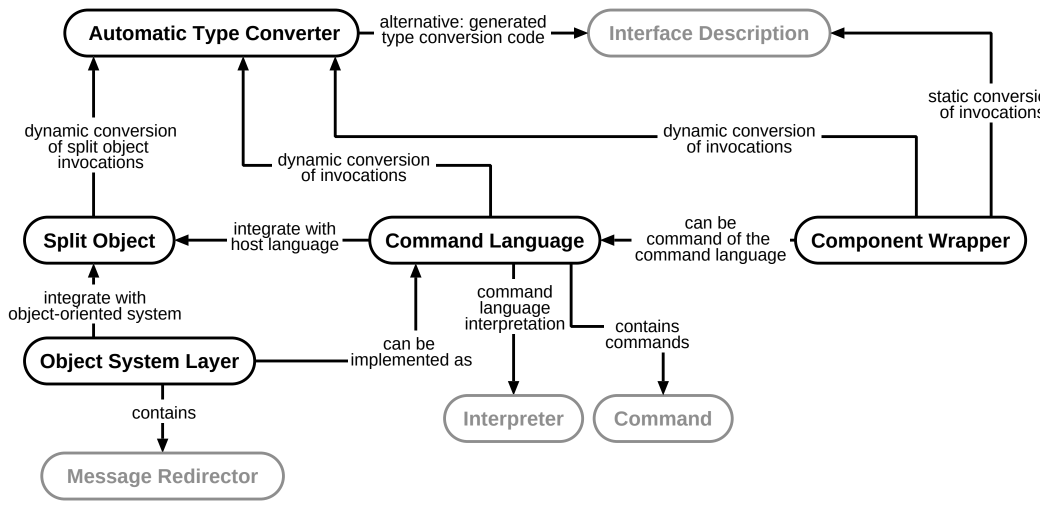
Automatic Type Converter
alternative: generated type conversion code
Interface Description
static conversion of invocations
dynamic conversion of split object invocations
dynamic conversion of invocations
dynamic conversion of invocations
Split Object
integrate with host language
Command Language
can be command of the command language
Component Wrapper
integrate with object-oriented system
command language interpretation
contains commands
Object System Layer
can be implemented as
contains
Interpreter
Command
Message Redirector

Figure 1: In the pattern map the most important relationships of the patterns are represented by labeled arrows.

a standard solution used in object-oriented and procedural systems: They encapsulate the implementation of an invocation and provide a generic, abstract invocation interface.

Problem **Using many COMMANDS without further support can be cumbersome in some cases, where the COMMANDS have to be composed in various different ways. Such situations occur frequently, when using COMMANDS for composing components or configuring a system, because in these situations a large number of different COMMANDS need to be composed. Runtime composition of the COMMANDS is not possible if the composition solution is hard-coded in static, compiled languages such as C, C++, or Java.**

Scenario Consider the following scenario: A system is built from a number of C components. Different customers require different components running in different configurations. The system needs to be composed and configured flexibly according to these customers' requirements. Rapid configuration to the customers' demands is needed. This is hard to achieve, however, because changing the system means programming in C, recompilation, and restart of the system.

Forces Composition in static, compiled languages, such as C, C++, or Java, is inflexible for situations where runtime behavior modification is required, because programming, recompilation, and restart are required for each change.

It is hard to implement simple, domain-specific languages, for instance for rapid configuration of software systems written in those languages, because this requires writing a parser, preprocessor, and interpreter for that language. Even if an existing parser, as for instance, an XML parser can be reused, still the mapping of language elements to implementations needs to be programmed by hand. For simple configuration options this is easily done, but for more complex languages that can handle behavior definitions, this can quickly result in a complex task.

The code of multiple, consecutive COMMAND invocations might be hard to read, just consider the following simple example:

```
if (expressionCommand.execute().toBoolean()) {
  result = command1.execute();
  command2.value = result;
  command2.execute();
}
```

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

Discussion A part of the COMMAND LANGUAGE is either an INTERPRETER [GHJV94] or an on-the-fly compiler. Note that often byte-code compilers are used for COMMAND LANGUAGES. A byte-code compiler can also be part of an INTERPRETER. From a language user's perspective all these alternatives look the same: The user operates in a language that is interpreted at runtime.

Almost all elements of a COMMAND LANGUAGE are COMMANDS, what means that a COMMAND LANGUAGE's syntax and grammar rules are usually quite simple. There are some additional syntax elements, as for instance grouping of instructions in blocks, substitutions, and operators:

- Grouping of instructions in blocks is required to build larger instruction sets, such as method bodies, lists of arguments, and so forth. Most often brackets are used to delimit grouping in blocks.
- Substitutions are required in a COMMAND LANGUAGE to hand over the results of one COMMAND to another. Other substitutions than command substitution are also possible, e.g. variable substitution or character substitution.
- Assignment operators are an alternative to command substitutions to hand over the results of one COMMAND to another. Also other operators can be implemented.
- The end of instructions has to be found during parsing. Usually either line ends or special characters (like "`;`") are used to mark an instruction end. A grouping can also mark an instruction end.

These COMMAND LANGUAGE elements are usually evaluated by the INTERPRETER before the COMMANDS are evaluated. In general, the INTERPRETER evaluates one instruction after another according to grouping and instruction end rules. Then it performs substitutions, if necessary. Finally it finds the COMMAND in the instruction and invokes it with the rest of the instruction as arguments.

Note that all further syntactic elements, as well as the semantics of the COMMAND LANGUAGE, must be well-defined, as in any other programming language. Finding the optimal syntax and semantics for a particular COMMAND LANGUAGE is beyond this pattern description because these issues depend on the concrete, domain-specific tasks the COMMAND LANGUAGE is used for. When the COMMAND LANGUAGE pattern is chosen as foundation of an integration architecture there is one important rule, however: Before building a COMMAND LANGUAGE from scratch, one should always consider to reuse an existing language that supports command execution, for instance, an extensible scripting language or an existing domain-specific language. It is often much easier to adapt an existing language infrastructure to the concrete requirements of a software system than building a language implementation from scratch.

Consider again the above simple example. Using a COMMAND LANGUAGE we can provide the dynamic expression by variable substitution (with '`$`') and pass the result of `command1` as an argument to `command2` (with '`[...]`'). These changes shorten the resulting code and make it much more readable:

```
if {$expression} {
  command2 [command1]
}
```

COMMAND LANGUAGE code is typically expressed as strings of the host language, in which the COMMAND LANGUAGE is implemented. From within the host language, code can be evaluated in the COMMAND LANGUAGE, and the results of these evaluations can be obtained. A central strength of the COMMAND LANGUAGE is that scripts can be (dynamically) assembled in strings and (also dynamically) evaluated later in time. In other words, we can write dynamically assembled and evaluated programs in host languages that do not support this feature natively. For instance, we can put the above script into a host language string, change this string at runtime, and evaluate it later in the program.

```
StringBuffer s = new StringBuffer("if {$expression} {command2 [command1]}");
...
// later change the script (here: append logging code)
script.append("\nputs \"evaluating expression: \$expression\"");
...
// later evalute the script
interpreter.eval(script.toString());
```

COMMANDS of a COMMAND LANGUAGE are typically implemented in the host language and bound to command names. The command names and implementations are registered in the INTERPRETER of the COMMAND LANGUAGE. When the INTERPRETER evaluates a COMMAND, it looks up the implementation, registered for a command name, and invokes it.

Usually all language elements of the COMMAND LANGUAGE are implemented in the same way as user-defined COMMANDS. For instance, control statements, such as `if` or `while`, are implemented as pre-defined COMMANDS. An `if`-COMMAND can be realized by first evaluating the expression given as first argument. If this expression returns `true`, the "then" part of the `if`-statement is evaluated in the INTERPRETER. An example is shown in Figure 2.

Sometimes interaction between code in the COMMAND LANGUAGE and the host language is necessary. Defining COMMANDS and using `eval`, as in the examples above, is one simple way of language integration. "Deeper" language integration is supported by the pattern SPLIT OBJECT. For all kinds of interaction, type conversion between the COMMAND LANGUAGE and the host language is necessary. This can be done by letting the user deal with type casting manually, or it can be automated using the pattern AUTOMATIC TYPE CONVERTER.

Scenario Resolved

Consider again the following scenario: When using a COMMAND LANGUAGE for integrating multiple C components, we can flexibly assemble these components and change the COMMAND LANGUAGE scripts even at runtime. Once a C component is integrated in the COMMAND

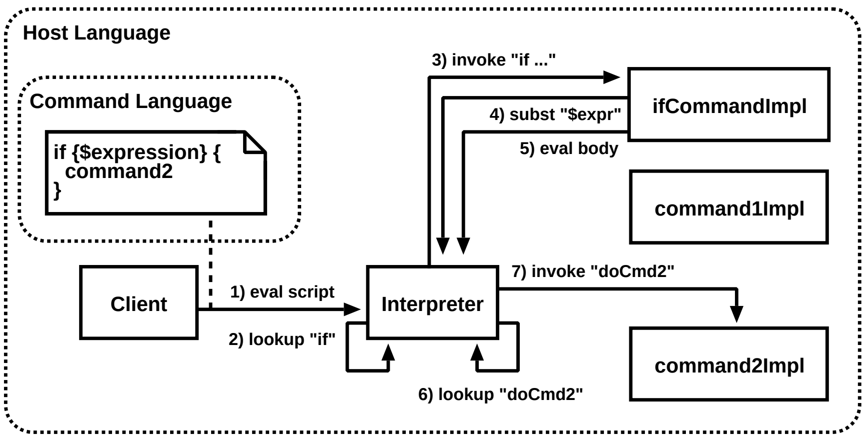

Figure 2: A script get executed by the command language interpreter

LANGUAGE, it can be used without C programming or recompilation. For instance, the three COMMANDS in the example in Figure 2 are implemented in the host language and can be freely composed using scripts. These scripts can be provided as stand-alone scripts to the COMMAND LANGUAGE'S INTERPRETER, read from files for instance. Alternatively, they can be evaluated as strings of the host language using the host language API of the COMMAND LANGUAGE.

Consequences COMMAND LANGUAGE composition is very flexible because COMMAND LANGUAGE scripts can be evaluated and changed at runtime. COMMAND LANGUAGES are thus well suited for rapid prototyping and for implementing domain-specific languages. Any data can be treated as program code of a COMMAND LANGUAGE.

A COMMAND LANGUAGE is primarily designed for being easily extensible with COMMAND implementations. Thus it is simple to extend the COMMAND LANGUAGE with new components, typically written in another language. This is usually done by registering COMMANDS with command names in the COMMAND LANGUAGE'S INTERPRETER. In many COMMAND LANGUAGES solutions exist for automating the integration of components into the COMMAND LANGUAGES as new COMMAND. How this is realized depends on the language features of the host language in which the COMMAND LANGUAGE is implemented. For instance, a wrapper generator can be used. Or, if supported, reflection can be used for performing a runtime lookup of the COMMAND implementation.

COMMAND composition in the host language is very efficient because COMMAND execution only requires one additional invocation in the host language. In contrast, a COMMAND LANGUAGE means a more severe overhead because it requires an additional dispatch of each invocation in the COMMAND LANGUAGE, as well as the interpretation of COMMAND LANGUAGE scripts.

Because the two languages have quite different tasks, each language can be designed for its particular task: The host language for writing efficient and reliable system components, and the COMMAND LANGUAGE for flexible component composition and other integration tasks. When using a COMMAND LANGUAGE together with a host languages, developers have to learn two languages. Often, this is not a great problem: Many COMMAND LANGUAGES are easy to learn because they a have a simple syntax and only a limited number of language elements is required for COMMAND composition and configuration.

Implementing a COMMAND LANGUAGE from scratch might be a substantial effort, depending on the language features required. Thus reusing an existing language infrastructure (such as an existing scripting language or domain-specific language) should be considered before implementing the language from scratch.

Using a COMMAND LANGUAGE consumes resources. Invocations that are sent through the COMMAND LANGUAGE have a weaker performance than directly invoked COMMANDS. The language implementation requires additional memory, too.

Known Uses Some known uses of the pattern are:

- Most scripting languages, such as Tcl, Python, and Perl are implemented as COMMAND LANGUAGES. That is, the language elements of the scripting language are implemented in another language; the languages named above are implemented in C. The INTERPRETER of the scripting language maps the command names in the scripts to the respective implementations in C. The scripting languages can thus be extended with new language elements implemented either in C or in the scripting language.
- A domain-specific language is a programming language tailored specifically for an application domain rather than providing a general purpose language. Many interpreted domain-specific languages are implemented as COMMAND LANGUAGES. This is because COMMAND LANGUAGES are easy to implement, easy to extend, or simply because another COMMAND LANGUAGE'S INTERPRETER is used for implementing the domain-specific language.
- The aspect-oriented framework of JAC [PSDF01] implements a COMMAND LANGUAGE for aspect composition and configuration. This way, operations of an aspect component can be provided as COMMAND implementations and invoked from the configuration file. Each aspect can define its own "little" configuration language. This aspect can be configured with custom parameters and behavior without the need for programming and re-compilation.

Component Wrapper

Context A system is composed from a number of black-box components. Component configurations and other component variations are handled by parameterization using the components' interfaces.

Problem **You require a flexible composition of the system's components. When using black-box components, component variations can only be introduced using parameterizations or exchange of the component. Parameters have to be foreseen by the component developers and thus are not well suited for unanticipated changes. Exchanging the component often means re-writing the component from scratch. For flexible composition, some solution is required that lets us deal with unanticipated changes for a component without sacrificing the black-box property.**

Scenario Consider the following scenario: A company has two related products, one implemented in C and one in C++. Both are monoliths, each consisting of a number of hard-wired subsystems. Consider the systems should be evolved into a product line, consisting of a number of products. It should be possible to flexibly assemble products according to customer demands. That is, it should be possible to compose new products from the existing system components. Often composition requires little extensions and changes to the existing components. The product line architecture should provide some concept for integrating components into products and for extending individual components in the context of a product with new state or behavior.

Forces

If the component's source code is available, one way to cope with unanticipated changes is to change the component itself. This is problematic because the black-box property of the component should not be violated: Black-box reuse eases understandability because the component can be reused without intimate knowledge of the component's internals. The component's internal implementation can even be replaced by another implementation, allowing us to get independent from implementation details. These are central reasons for using the component-based approach – and thus should not be sacrificed without need.

Components – especially third-party components – change over the time. Even though the black-box abstraction aims at stable interfaces, often the interfaces change. The client's developer should be able to cope with these changes without a necessity to seek for component accesses through the whole client's source code.

In many situations, one and the same client version has to deal with several component versions. Then adaptations to interfaces and to relevant internal implementation details have to take place within the same client implementation. With techniques such as preprocessor directives or if/switch control structures this may result in hard to read code.

Usually a client of a black-box component should be independent of the internal implementation of the component. But sometimes implementation details of the component implementation, such as access to shared resources, performance issues, and memory consumption, do matter and need to be accessed somehow.

Solution

Let the access point to a component, integrated into a system, be a first-class object of the programming language. Indirection by a COMPONENT WRAPPER object gives the application a central, white-box access point to the component. Here, the component access can be customized without interfering with the client or the component implementation. Because all components are integrated in the same way, a variation point for white-box extension by component's clients is provided for each black-box component in a system.

Discussion

The COMPONENT WRAPPER provides access to a component using the object abstraction of the component's client. A simple case is that both the component implementation and the component client use the same object abstraction; however, this is not always the case. In fact, a major goal of the COMPONENT WRAPPER pattern is to integrate components, written in different paradigms and languages, using the same integration style. To reach this goal, the COMPONENT WRAPPER pattern can use some other patterns within its internal implementation (there are many other integration styles though):

- In the simple case that another object-oriented component is integrated, the COMPONENT WRAPPER can use the PROXY pattern [GHJV94]. A PROXY is a placeholder object for another object offering the same interface. Note that a COMPONENT WRAPPER interface can also be different to the component's interface or only provide an excerpt of it, as for instance an "exported" interface. If two different object-oriented languages with slightly different object and type concepts are integrated, adaptations and type conversions can take place on the COMPONENT WRAPPER.

- When we want to integrate a procedural paradigm component into an object-oriented implementation, we apply the WRAPPER FACADE pattern [SSRB00]. It uses one or more objects to represent a procedural component. The wrapper object forwards object-oriented messages to the procedures (or functions).

- If a COMMAND LANGUAGE is also used, the COMPONENT WRAPPERS can be exposed as COMMANDS of the COMMAND LANGUAGE. As a consequence, COMPONENT WRAPPERS can then be flexibly assembled using COMMAND LANGUAGE scripts. That is, two variability mechanisms are combined here: The COMPONENT WRAPPER allows for variation of the individual components and their integration in the system; the COMMAND LANGUAGE scripts allow for flexibly composing multiple COMPONENT WRAPPERS in the system. The COMPONENT WRAPPER is exposing the component's export interface as operations of the COMMAND, one for each exported functionality of the component. The arguments of the COMMAND in the COMMAND LANGUAGE script are used to dispatch the correct COMPONENT WRAPPER operation.

In each of the three variants, the COMPONENT WRAPPER can expose the same interface as the wrapped component (then it is a PROXY), or a different one. When changes to types of the interface are necessary during an invocation by a COMPONENT WRAPPER, the COMPONENT WRAPPER can either contain the conversion code or use an AUTOMATIC TYPE CONVERTER for the conversion.

COMPONENT WRAPPERS are used to provide a uniform way of wrapping black-box components. The component client does not know about the style of wrapping or other implementation details. Moreover, the indirection to a COMPONENT WRAPPER object provides an accessible white-box on client-side. This placeholder provides a central access point to the component. DECORATORS [GHJV94], ADAPTERS [GHJV94], MESSAGE INTERCEPTORS [Zdu03b], and other kinds of extensions, variations, or customizations optionally can be added to the COMPONENT WRAPPER. Only changes that necessarily have to interfere with the component's internals cannot be handled this way.

COMPONENT WRAPPERS can easily be generated, if the component's interface is electronically documented with an INTERFACE DESCRIPTION [VKZ04]. The default implementation of such generated COMPONENT WRAPPERS just forwards the invocation to the component, but the client is able to extend the generated code. Using techniques, such subclassing, polymorhism, or MESSAGE INTERCEPTORS [Zdu03b], we can avoid that re-generated code overwrites client extensions of the COMPONENT WRAPPER.

Using COMPONENT WRAPPERS we can modularize the aspect "integration of another component". Thus COMPONENT WRAPPERS are well suited to be applied together with an aspect-oriented approach [KLM$^+$97]. That is, an aspect composition framework can be used to weave the aspect represented by the COMPONENT WRAPPER into the system.

Scenario Resolved

Consider again the following scenario: Two monolithic products should be evolved into a product line architecture. This can be done in an incremental fashion using COMPONENT WRAPPERS: They are used to decouple the components of the legacy systems step by step. Figure 3 shows that COMPONENT WRAPPERS are implemented as part of the product line, and instantiated by a new product. There is one COMPONENT WRAPPER for each of the export

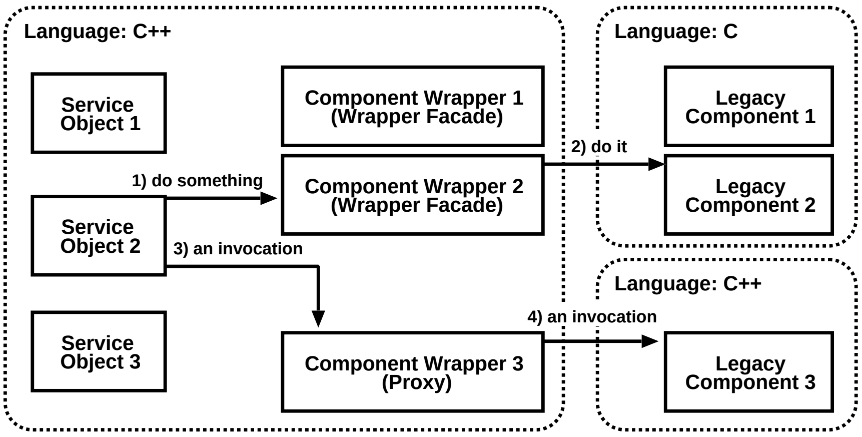

Figure 3: A number of component wrappers are integrated into a C++/C system: Some are wrapper facades, some are proxies

interfaces of the products. For the C product the pattern WRAPPER FACADE is used, for the C++ product we use a PROXY. In the example we can see that one COMPONENT WRAPPER performs an interface adaptation; the PROXY only forwards the invocation. On the COMPONENT WRAPPERS individual products, composed from the existing components, can add behavior or state to the components.

Consequences Using COMPONENT WRAPPERS customizability of black-box components is enhanced. Different styles of wrapping are conceptually integrated from a component client's perspective. Application developers can flexibly adapt and change component interfaces at a central access point. For instance, slight version changes in component version can be hidden by the COMPONENT WRAPPERS so that clients do not have to contain the adaptation code (such as `#IFDEF` directives). In other words, COMPONENT WRAPPERS reduce tangled code caused by component integration, composition, and configuration.

Using COMPONENT WRAPPERS an external component is used as an internal, first-class entity of the object system. That is, once integrated, all external and internal components look the same for component clients. The component use is decoupled from the internal realization. Therefore, the component itself is exchangeable with another implementation.

The indirections to the COMPONENT WRAPPER reduce performance slightly. More classes and flexibility hooks can result in a higher complexity of the application and thus reduce understandability. COMPONENT WRAPPERS introduce additional code. Thus there is more code to test and understand. Due to the indirection and the additional code, system's using COMPONENT WRAPPERS might be slightly harder to debug.

Known Uses Some known uses of the pattern are:

- In [GZ02] we present a case study of a reengineering project for a document archive system implemented in C. We use COMPONENT WRAPPERS for migrating the existing C implementation into a component architecture in a stepwise manner.
- SWIG [Swi03] is a wrapper generator that can generate COMPONENT WRAPPERS for C and C++ components in various other languages. SWIG uses language bindings and header files as INTERFACE DESCRIPTIONS.

- VCF [OGJ03] integrates components from various Java component models, namely COM+, CORBA, EJBs, and Java Beans. A FACADE provides a common denominator interface of the technologies to clients. For integration of the component models, the approach uses a plug-in architecture that is based on a number of COMPONENT WRAPPERS.

Object System Layer

Context Parts of a system implementation are realized in a language that does not support object-orientation, such as C, Tcl, or Cobol. Or the implementation is realized in an object system that does not support desired object-oriented techniques. For instance, C++ or Object Cobol do not implement reflection or interception techniques.

Problem **Suppose you want to design with object-oriented techniques and use the benefits of advanced object-oriented concepts, but you are faced with target programming languages that are non-object-oriented, or with legacy systems that cannot quickly be rewritten, or with target object systems that are not powerful enough or not properly integrated with other used object systems, say, when COM or CORBA is used. But the target language is chosen for important technical or social reasons, such as integrating with legacy software, reusing knowledge of existing developers, and customer demands, so it cannot be changed. One solution is to translate the design into a non-object-oriented design, and then to implement that design. If this mapping is manual then it will be error-prone and will have to be constantly redone as the requirements change.**

Scenario Consider the following scenario: Legacy applications are often written in procedural languages, such as C or Cobol. A complete migration of a system to a new object-oriented language often makes the integration with the existing legacy components difficult and forces a re-implementation of several well functioning parts. The costs of such an evolution are often too high to consider the complete migration to object technology at all. Most often the costs are even very hard to estimate in advance.

Forces There are a number of situations in which we require some integration with an object-system, as for instance:

- Many modern systems are based on object-oriented concepts, but still legacy systems offering no support for object-orientation need to be supported.
- When two or more different object-oriented languages need to be integrated, we can use COMPONENT WRAPPERS for integration. However, we should avoid to implement this integration for each individual COMPONENT WRAPPER on its own.
- The need for integration of object concepts might also occur in only one object-oriented language when foreign object concepts need to be integrated. For instance the CORBA or COM object systems are not exactly the same as the object systems of object-oriented languages, such as C++ or Java, in which they are used. This problem arises for all kinds of technologies that introduce object concepts, as for instance distributed object systems, application servers, transaction monitors, and relational and object-oriented databases.

- Many applications use an object system of a mainstream programming language, such as C++, Java, or Object Cobol. Those object systems do only implement standard object-oriented techniques. Thus extensions of object concepts, such as interception techniques, reflection and introspection, role concepts, or aspect-orientation are not provided natively.

To integrate object-oriented systems with non-object-oriented systems, or to integrate different object systems, we require some mapping of the language concepts. Such mappings should be reusable so that a programmer does not require to perform the integration over and over again.

Solution

Build or use an object system as a language extension in the host language and then implement the design on top of this OBJECT SYSTEM LAYER. Provide a well-defined interface to components that are non-object-oriented or implemented in other object systems. Make these components accessible through the OBJECT SYSTEM LAYER, and then the components can be treated as black-boxes. The OBJECT SYSTEM LAYER acts as a layer of indirection for applying changes centrally.

Discussion

An OBJECT SYSTEM LAYER requires some dispatch mechanism that translates invocations in the host language into invocations to the objects of the OBJECT SYSTEM LAYER. This can be done using a MESSAGE REDIRECTOR [GNZ01]. It acts as a FACADE [GHJV94] to the whole OBJECT SYSTEM LAYER and is not bypassed. For instance, the MESSAGE REDIRECTOR invocations might be string-based messages, which are translated into the respective invocations.

The OBJECT SYSTEM LAYER can be part of a COMMAND LANGUAGE, if the COMMAND LANGUAGE implements an object system. Then the OBJECT SYSTEM LAYER's MESSAGE REDIRECTOR is part of the COMMAND LANGUAGE's INTERPRETER. Once a script is parsed, the INTERPRETER uses a MESSAGE REDIRECTOR to indirect the invocation to the implementation of the COMMAND LANGUAGE element. A typical integration of the patterns OBJECT SYSTEM LAYER and COMMAND LANGUAGE uses one object per COMMAND, and the command name is used as an object ID for the OBJECT SYSTEM LAYER object.

The SPLIT OBJECT pattern is used, if the OBJECT SYSTEM LAYER needs to be integrated with a host language object system. Using COMPONENT WRAPPERS foreign paradigm elements, such as procedural APIs, can be integrated into the OBJECT SYSTEM LAYER's object system.

Besides the conceptual integration with host language concepts, an OBJECT SYSTEM LAYER also must deal with other potential mismatches of the host language and the OBJECT SYSTEM LAYER's model. For instance, mismatches in the threading models or mismatches in transaction contexts must be resolved.

The OBJECT SYSTEM LAYER can be used to introduce extensions of the object concept of an object-oriented language. Typical examples are the introduction of runtime dynamic object and class concepts such as TYPE OBJECTS [JW98], role concepts, or MESSAGE INTERCEPTORS [Zdu03b].

Many technologies integrated into a system, implement an OBJECT SYSTEM LAYER, as for instance distributed object systems, application servers, transaction monitors, and databases. If needed, an OBJECT SYSTEM LAYER can also be introduced as a superset of (a part of) two or more object concepts to integrate them. For instance, the OBJECT SYSTEM LAYERS of technologies that are implemented in multiple languages, such as distributed object frameworks, are primarily used for integrating the object concepts of these languages.

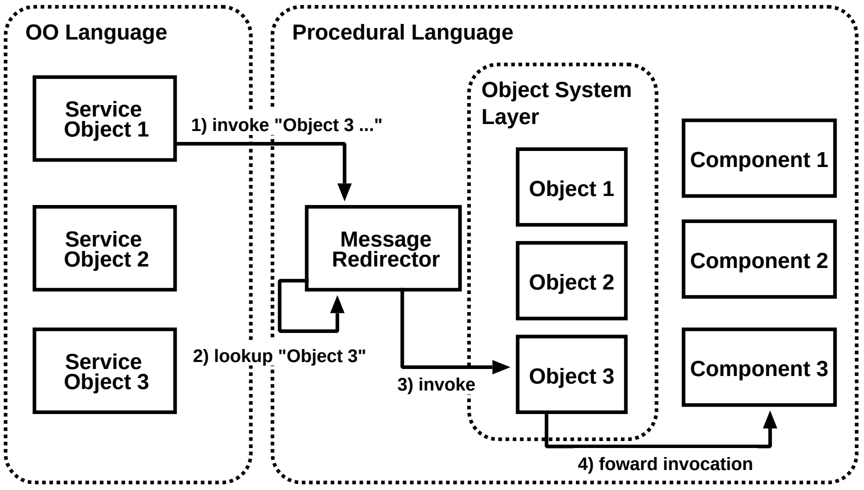

Figure 4: An object system layer is embedded in a procedural language

Scenario Resolved Consider again the following scenario: Procedural components should be composed using object-oriented concepts. An OBJECT SYSTEM LAYER can be used for this purpose, as depicted in Figure 4. This has the benefit that we do not have to translate the system into an object-oriented language completely. Instead existing components can be incrementally evolved; others can stay in the procedural language and be integrated into the OBJECT SYSTEM LAYER using COMPONENT WRAPPERS.

Consequences When using an OBJECT SYSTEM LAYER, the flexibility of the application can be improved, since the OBJECT SYSTEM LAYER allows us to introduce variation points and to implement high-level language constructs, such as interception, reflection, and adaptation techniques.

The complexity of the application can be reduced, because – in scenarios where they are really needed – more high-level concepts, such as roles, interceptors, or aspects can simplify the program code and avoid complex workarounds.

Complexity of the application can also rise, if the client has to maintain the OBJECT SYSTEM LAYER. Then issues such as garbage collection, object destruction, relationship dynamics, reflection, etc. have to be programmed by hand. But this problem can be avoided by using an existing OBJECT SYSTEM LAYER as a library.

Performance can be decreased due to additional indirections.

The OBJECT SYSTEM LAYER's conventions and interfaces have to learned by the developers in addition to the system's APIs.

Known Uses Some known uses of the pattern are:

- Object-oriented scripting languages implemented in C, such as Python or XOTcl [NZ00], provide an OBJECT SYSTEM LAYER for C. These languages provide a C API – that means the object models can be accessed and used from C for using object-oriented abstractions in C programs.

- The Redland RDF library [Bec04] implements a simple OBJECT SYSTEM LAYER for C to use object-oriented abstractions for RDF nodes, models, streams, and other elements of the library. The library implements classes, object IDs, constructors, and destructors.

- Procedural implementations of object-oriented middleware, such as C or Tcl implementations of CORBA or Web Services, provide the object abstractions of the middleware in the procedural language. Thus in the distributed context these implementations provide an OBJECT SYSTEM LAYER.

Automatic Type Converter

Context Languages or systems supporting different types are used in one system.

Problem **When two different type systems need to be integrated, it is necessary to convert corresponding types. An INTERFACE DESCRIPTION can be used to describe the type differences, and the corresponding type conversion code can be generated. Conversion code for each particular invocation can only be generated, however, if the signatures of all operations are known during generation. Sometimes operation signatures are not known before runtime. In many integration situations it is also necessary that it is easy to add new type bindings rapidly.**

Scenarios Consider the following scenarios:

- In Web Services frameworks remote invocations and other messages are transferred using SOAP [BEK+00] as a XML-based payload format. In SOAP, invocations containing parameter and return types are encoded in string format. To be interpreted by a programming language the SOAP messages must be converted to programming language native formats. From an INTERFACE DESCRIPTION we can generate a client proxy and server stub that perform the type conversion. This, however, does only work, if the types are known before runtime; for dynamic deployment of services we basically have two choices: We can generate a server stub while the system runs or and perform type conversions at runtime.

- A COMMAND LANGUAGE is embedded in a statically typed language. The COMMAND LANGUAGE is given in form of scripts that are interpreted at runtime. That is, all types to be accessed by the COMMAND LANGUAGE need to be dynamically convertible to and from strings. To ease type integration, it should be simple to integrate new types.

Forces Different languages support different types that cannot be exchanged directly. This problem also arises for other systems supporting types – not only for programming languages. For instance, most distributed object frameworks support their own type system to integrate across languages or platforms. Systems supporting dynamic types – for instance using the pattern TYPE OBJECT [JW98] – require some way to cast the dynamic types into other dynamic types or static types of the host language. Some languages and typed systems only support canonical types, such as strings. To integrate them with a strongly typed system, we need to convert the types to the canonical format and vice versa.

When an INTERFACE DESCRIPTION of all operation signatures exists, an option is to generate the type conversion code. This option is for instance often chosen in the proxies and stubs generated by many distributed object frameworks. However, this does not work for more dynamic or complex type conversion situations. For instance, generation is not as well suited for dynamic typing or runtime deployment because here the operation signatures are not known before runtime of the system. Note that runtime generation of conversion code is possible – it is just more complex than writing simple wrappers.

Also, in many cases, it is necessary to provide a very simple kind of type conversion because it should be easily extensible: Only the type binding and the conversion code should be needed to be specified.

Solution **Provide an AUTOMATIC TYPE CONVERTER as a means to convert an instance of a particular type supported by the AUTOMATIC TYPE CONVERTER to a particular target type, and vice versa. The AUTOMATIC TYPE CONVERTER is either extensible with new type bindings for one-to-one transformations, or converts every type to a canonical format, such as strings, and can convert the canonical format to any supported type. If no user-defined type binding is found, the AUTOMATIC TYPE CONVERTER should execute a default conversion routine.**

Discussion In general, an AUTOMATIC TYPE CONVERTER should be able to deal with all kinds of conversions that occur at runtime because manual intervention is not possible at runtime and a conversion that fails leads to a runtime error. That is, all kinds of AUTOMATIC TYPE CONVERTERS should always provide a default conversion routine that takes place when no custom type binding is defined. Hooks for type conversion are provided so that users can extend the type converter with additional conversions.

There are two main variants of AUTOMATIC TYPE CONVERTERS:

- There might be a canonical format, such as strings, `void*` in C/C++, `Object` in Java, supported by the AUTOMATIC TYPE CONVERTER and a number of other types. The AUTOMATIC TYPE CONVERTER can convert any type to the canonical format and the canonical format to any type. This way, any type can be converted into any other type by a two-step conversion. Note that it is the developer's responsibility to provide sensible canonical representations of the various types which allow for conversion without information loss and which are efficient. Also, it is the developers burden to request only sensible conversions. For instance, converting a whole database into a in-memory string representation is problematic; it might be better to convert a database handle only. The system must be designed in a way so that continuous back-and-forth conversions are avoided, where possible.

- An alternative to using a canonical format is to use direct, one-to-one conversion. This is usually faster because only one conversion is required. The drawback of this variant is that we might have to write more converters: If each type can be mapped to each other type, we have to write $N*M$ converters, instead of N converters needed to support a canonical format. Therefore, one-to-one conversion is often chosen if there is only one target type for each type. A second reason for using one-to-one conversion is that specialties of the individual types must be considered during conversion, for instance, because information would get lost during conversion to a canonical format. Finally, a third reason for choosing one-to-one conversion might be the better performance because conversion to/from the intermediate canonical representation can be avoided.

An AUTOMATIC TYPE CONVERTER can also be used for more than two types. Thus, for instance, it can also be used to implement polymorphism for TYPE OBJECTS and other dynamic typing solutions. The AUTOMATIC TYPE CONVERTER can convert an object to all types, supported by its classes, and delivers the casted type, as requested by the client. This is for instance useful if an AUTOMATIC TYPE CONVERTER is combined with an OBJECT SYSTEM LAYER or SPLIT OBJECTS.

A COMMAND LANGUAGE implemented in a host language usually requires an AUTOMATIC TYPE CONVERTER. In this scenario, the canonical format variant is often supported because the COMMAND LANGUAGE scripts are represented as strings, and these strings can be used as a canonical format. Thus the AUTOMATIC TYPE CONVERTER is used as a mechanism to exchange data, specify operation parameters, and receive operation results between a COMMAND LANGUAGE and its host language.

An AUTOMATIC TYPE CONVERTER is a flexible alternative to type conversion code generated from an INTERFACE DESCRIPTION. In many cases, generated type conversion code uses the AUTOMATIC TYPE CONVERTER internally. An AUTOMATIC TYPE CONVERTER can also use INTERFACE DESCRIPTIONS as an aid in the type conversion decision.

The pattern CONTENT CONVERTER [VZ02] is a similar pattern describing content integration – it also can be used with a canonical format or using one-to-one integration. If the content of a CONTENT CONVERTER is typed, an AUTOMATIC TYPE CONVERTER can be used as part of a CONTENT CONVERTER used for the type conversion task.

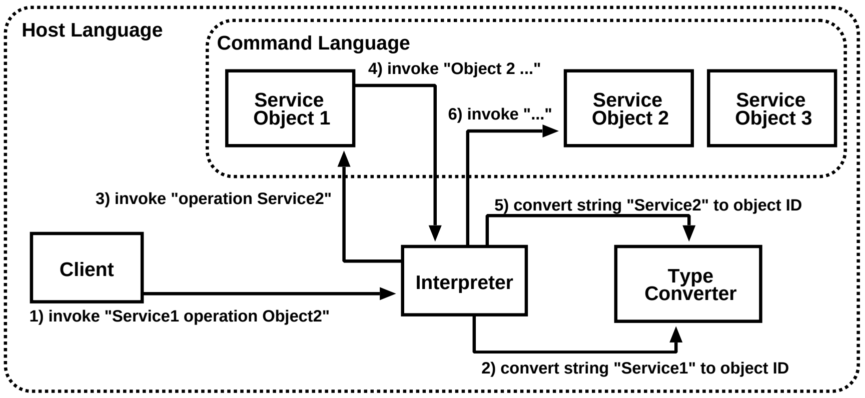
Host Language
Command Language
4) invoke "Object 2 ..."
Service Object 1
6) invoke "..."
Service Object 2
Service Object 3
3) invoke "operation Service2"
5) convert string "Service2" to object ID
Client
Interpreter
Type Converter
1) invoke "Service1 operation Object2"
2) convert string "Service1" to object ID

Figure 5: A host language client invokes an command language object and the interpreter performs automatic type conversions

Scenarios Resolved

Consider again the following scenarios:

- In a Web Service framework we usually can map every programming language type to one type in the INTERFACE DESCRIPTION. Thus here we can use a one-to-one converter between INTERFACE DESCRIPTION type, transmitted over the network, and the type of the programming language.

- A COMMAND LANGUAGE needs to be integrated with a statically typed language. The COMMAND LANGUAGE uses scripts consisting of strings. Thus we need a converter for each host language type to strings and from strings to each host language type. In Figure 5 a solution using a string representation as a canonical format is sketched. Each AUTOMATIC TYPE CONVERTER object internally has two representations: A string representation and an internal representation containing the target type of the statically typed host language. The AUTOMATIC TYPE CONVERTER class provides an operation for converting the internal representation to strings, an operation for converting strings to the internal representation, getter/setter operations for both representations, and operations for discarding one of the two representations. At each point in time, at least one of the two representations must be valid.

Consequences An AUTOMATIC TYPE CONVERTER is in many scenarios slower than a solution generated from an INTERFACE DESCRIPTION because it performs the conversion at runtime, whereas the generated solution only performs a single cast. The benefit of an AUTOMATIC TYPE CONVERTER is that it is more flexible than generated solutions: This way dynamic typing can be supported. Type mappings can be changed at runtime. Also, the type mappings in an AUTOMATIC TYPE CONVERTER are easy to extend.

Known Uses Some known uses of the pattern are:

- The scripting language Tcl uses an AUTOMATIC TYPE CONVERTER for converting the string-based scripts to native types in C and vice versa. Strings are used as a canonical format.
- IONA Artix is a Web Service framework that supports conversion between different protocols (HTTP, IIOP, Websphere MQ, Tuxedo, etc.) and payload formats (SOAP, Tuxedo's FML, GIOP, etc.). To increase conversion performance, Artix uses one-to-one converters. An AUTOMATIC TYPE CONVERTER for each converter is needed to recognize the types in one payload format and translate them into the target format.
- Apache Axis is a Web Service framework that uses AUTOMATIC TYPE CONVERTERS for one-to-one conversion between programming language types in Java and the payload format SOAP.

Split Object

Context Two languages need to be used together in one system.

Problem **To integrate two languages one can write simple wrappers that allow for invocations in the other language. However, pure wrapping poses some problems in more complex language integration situations. A wrapper provides only a "shallow" interface into a system, and it does not reflect further semantics of the two languages. Examples of such semantics are class hierarchies or delegation relationships. Further, a wrapper does not allow one to introspect the system's structure. The logical object identity between wrapper and its wrappees is not explicit. Complex wrappers that are implemented manually are hard to maintain.**

Scenario Consider the following scenario: One language embeds another language, such as a COMMAND LANGUAGE that is embedded in a host language for configuring host language objects. That is, the COMMAND LANGUAGE requires access to its host language, and vice versa. Here "access" means, for instance, performing lookups, invocations, creations, and destructions of objects and methods.

Forces A main driving force for considering deeper language integration than COMPONENT WRAPPERS it that it is cumbersome to write wrappers for each and every element that requires language integration. To a certain degree, wrappers can be generated, but still we would require custom code for each language integration situation. Instead it should be possible to reuse language integration code. Automation and reuse would also foster maintainability: In general, complex wrapping structures can quickly get quite complex and thus hard to maintain. Especially scattering wrapping code across the functional code should be avoided.

Simple wrappers are usually realized by a single indirection, and thus they are relatively fast and consume only little extra memory. A deeper language integration needs to be compared to simpler wrapper solutions with regard to resource consumption.

Solution **A SPLIT OBJECT is an object that physically exists as an instance in two languages, but logically it is treated like one, single instance. Both SPLIT OBJECT halfs can delegate invocations to the other half. The SPLIT OBJECT halfs mimic the user-defined class hierarchy of the counterpart, variables are automatically traced or shared, and methods can be wrapped. Depending on the language features of the two languages, these functionalities can either be implemented by extending the language's dispatch process, using reflection, or using program generation.**

Discussion There are different ways how a SPLIT OBJECT half can communicate with its counterpart in the other language:

- A method can be provided, for instance by a superclass of all SPLIT OBJECTS, that allows for sending an invocation to the other half. The invocation parameters are usually provided as arguments using a generic type (such as strings or `Object` in Java) and converted using an AUTOMATIC TYPE CONVERTER. Depending on the language features, the method implementation on the other split object half can be looked up using reflection, using a dynamic dispatch mechanism, using a registration in a method table (provided by the developer), or using a generated method table.

- A wrapper generator can generate a method on a split object half for each exported method of the other split object half. This generated method is a simple forwarder to the other split object half. The invocation parameters can be provided using the correct types, required by the counterpart, if possible, or an AUTOMATIC TYPE CONVERTER is needed for conversion. This variant is very efficient because it requires no dynamic lookup.

- If the language's dispatch mechanism can be extended and method dispatch is performed at runtime, we can implement an automatic forwarding mechanism. Whenever a method cannot be found for the split object half, the "method not found" error is not raised directly, but it is first tried to find the method on the other split object half. If this succeeds, the method is invoked there, otherwise the error is raised. In this variant, the same means for method lookup as in the first alternative can be used.

- In dynamic languages (e.g. a COMMAND LANGUAGE), wrapper methods can also be generated at runtime. Again, these methods are simple forwarders, as in the second alternative, but the means for method lookup, described in the first alternative, are used.

Note that only in the first variant clients recognize that the object does not itself implement the method; in the following three variants the client can use the same invocation style as used for methods implemented in the same language.

In many cases, split object halfs will mimic the user-defined class hierarchy of the other half. For instance, when you generate forwarder methods to the other split object half, you face the problem of name clashes: Methods on different classes of one object might have the same name. This problem does not arise, if every class for which a forwarder method is generated, exists within both languages. For this to work, to a certain extent, we need to integrate the class concepts of the two languages. For instance, if one language supports multiple inheritance, and the other not, we need to simulate multiple inheritance in the other language. To mimic a class hierarchy in a procedural language or integrate different class concepts in two object-oriented languages, you might consider implementing a simple OBJECT SYSTEM LAYER.

From the COMMAND LANGUAGE we can automatically forward invocations into the host language. In turn, ordinary host language invocations bypass the SPLIT OBJECT in the COMMAND LANGUAGE. Thus, from within the host language, we have to use the INTERPRETER'S `eval` method to access a SPLIT OBJECT. To avoid this additional invocation style, the pattern HOOK INJECTOR [Zdu03b] can be used to replace host language invocations with indirections to SPLIT OBJECTS.

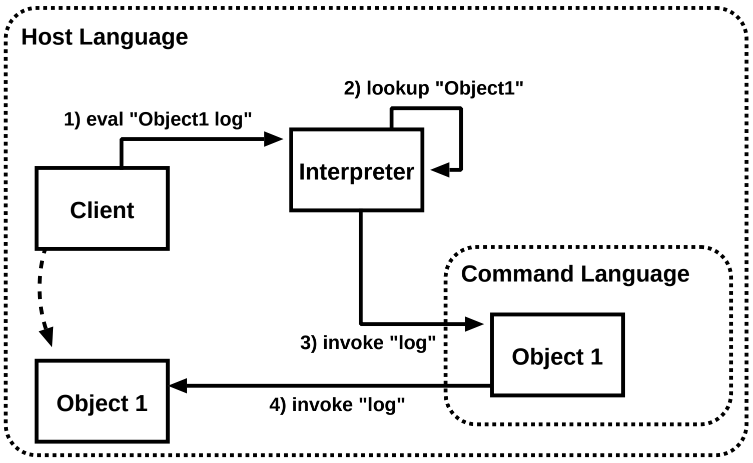

Figure 6: Invoking a split object through the command language

Scenario Resolved Consider again the following scenario: When integrating a COMMAND LANGUAGE with a host language, we can provide a method for invoking the COMMAND LANGUAGE from the host language. The COMMAND LANGUAGE object, in turn, provides an automatic forwarding mechanism to invoke the host language half of a SPLIT OBJECT. As an example, Figure 6 shows a host language client that needs to invoke a host language object `Object1`. This object is a SPLIT OBJECT: Instead of invoking this object directly, the counterpart in the COMMAND LANGUAGE is invoked first, which forwards the invocation back into the host language. This way the COMMAND LANGUAGE can intercept the invocation.

Consequences SPLIT OBJECTS can be used to deeply integrate two object systems. Concepts realized in one object system can be used from within the other object system. The language integration code in the SPLIT OBJECT implementation can be reused. Thus language integration is easier to apply and the code is easier to maintain than integration code based on COMPONENT WRAPPERS only. SPLIT OBJECTS can be used as a model to automate wrapper generation for two languages.

Working with two languages might make debugging more difficult because you have align the debugging information in the two languages and maybe even integrate debugging tools. On the other hand, SPLIT OBJECTS can even be used to provide debugging information for other languages (see [Zdu04] for examples).

SPLIT OBJECTS pose a memory and performance overhead. This overhead can be minimized by not using dynamic lookup and dispatch mechanisms, but generative techniques instead. Generated code is hard to change at runtime (if this is possible at all); that is, using generative techniques means less flexibility.

Known Uses Some known uses of the pattern are:

- The network simulator [UCB00a] and Open Mash [Ope00] are two projects that are implemented in C++ and use OTcl [WL95] for configuring and customizing the C++ applications. The integration of the two languages is performed by a SPLIT OBJECT solution called Tclcl [UCB00b].
- In [Zdu04] it is demonstrated how to use SPLIT OBJECTS for dynamically performing maintenance tasks, such as component testing, feature tracing, and variation and configuration management. For instance, the language Frag [Zdu03a] is used to configure and compose AspectJ [KHH$^+$01] aspects dynamically to perform these tasks in Java.
- The Redland RDF library contains an C-based OBJECT SYSTEM LAYER. The Perl interface to that system uses a hand-build SPLIT OBJECT solution: When a Perl wrapper is created, the respective C-based object is created as well – thus object IDs are mapped one to one. Similarly, the C destructors are automatically invoked in the correct order upon destruction of the Perl wrapper object.

Known Uses and Technology Projections

In this section we discuss some known uses and examples for the patterns in an integrated manner. We rather concentrate on a few interesting technology projections than discussing all known uses; there are simply too many. In this section, we present examples from the areas of distributed object frameworks, reengineering, scripting languages, and dynamic aspect configurations. There are many other systems in each of these areas using some of the patterns, described in this paper.

Apache Axis

We want to discuss the Web service framework Apache Axis as an example of a distributed object framework. In many other distributed object frameworks similar concerns occur and similar solutions are applied.

The MARSHALLER [VKZ04] of the Web service framework needs to convert the types of remote invocations into strings to be transported across the network using the SOAP protocol, and on server side it needs to convert the strings into the respective types again. In Axis the

XML Schema Data (XSD) types are used, and WSDL files are used as an INTERFACE DESCRIPTION. Thus Axis only requires one-to-one conversion between programming language types and the string representation of the XSD types. This is done using an AUTOMATIC TYPE CONVERTER. This way type conversion is flexibly extensible with new type converters.

In Axis a type converter consists of a factory for a serializer and de-serializer, as well as an implementation of serializer and de-serializer. Besides primitive types which can use a simple converter because type conversion is supported by Java, Axis can convert many more complex types, such as arrays, dates, calendars, DOM documents, object references, vectors, beans, and others. In a class containing the type bindings, these AUTOMATIC TYPE CONVERTERS are mapped to the XSD types. According to the INTERFACE DESCRIPTION, Axis can lookup the Java type converter corresponding to an XSD type and perform the conversion automatically.

A Web service framework usually supports heterogeneity of backends implementing the service. For instance, a service might be implemented as a Java class, a procedural library, an EJB component, a COM component, and many others. Axis provides a "provider" abstraction. One provider class for each type of backend service is provided. The provider can be dynamically configured for particular invocations, and the framework can be extended with new providers for new types. This is a variant of the pattern COMPONENT WRAPPER: Each backend component type can be flexibly integrated, and we can manipulate the mapping by either providing new provider types or by configuring the provider objects.

A distributed object framework is an implementation of a canonical object system, often implemented in many languages. Thus is can be used as an OBJECT SYSTEM LAYER to provide objectified access to non-object-oriented languages, as well as a unified access to the object systems of the language supported. The object concepts of this OBJECT SYSTEM LAYER are usually described by a language for INTERFACE DESCRIPTIONS, such as WSDL for Web Services and IDL for CORBA.

Reengineering a Document Archive System

In [GZ02] we present a case study of reengineering a large-scale documented archive system, implemented in C, to an object-oriented architecture. An important concern was incremental evolution of the system. Here, the pattern COMPONENT WRAPPER played a central role: The idea was to start off with simplistic wrappers, in particular WRAPPER FACADES for the original procedural components – these were more or less similar to the original subsystems of the system. As the reengineering project continues, different component versions have to be supported, interface changes have to be handled, and components with other wrapper styles have to be integrated as well. COMPONENT WRAPPERS provide a way of unifying wrapping styles and handling extensions or adaptations.

To objectify the system, we applied the pattern OBJECT SYSTEM LAYER. A simple object system was provided with which the components can be treated in an object-oriented manner and easily integrated with the object systems of distributed object systems (in [GZ02]: CORBA).

New requirements impose changes on the wrappers constantly. Such extensions or adaptations can be transparently handled by MESSAGE INTERCEPTORS [Zdu03b]. Here, the pattern OBJECT SYSTEM LAYER is beneficial again, because the MESSAGE REDIRECTOR of the OBJECT SYSTEM LAYER can be used to apply the interceptors. It can also trigger an AUTOMATIC TYPE

CONVERTER, where necessary.

TclCL and XOTcl/SWIG

TclCL [UCB00b] is an integration of OTcl [WL95] and C++. It utilizes SPLIT OBJECTS for language integration. TclCL is used in:

- Mash [Ope00] is a streaming media toolkit for distributed collaboration applications based on the Internet Mbone tools and protocols.
- The Network Simulator (NS) [UCB00a] supports network simulation including TCP, routing, multicast, network emulation, and animation.

In both projects, TclCL is applied to utilize the two language concept: The COMMAND LANGUAGE OTcl is used for high-level tasks, such as configuration, testing, and others, and C++ is used for the efficient implementation of the core tasks as C++ COMMANDS. The user-defined OTcl classes mimic the C++ class hierarchy. This is mainly done because the C++ variables/methods can be dynamically registered for OTcl in the C++ constructors of the respective classes. The OTcl counterpart remembers the capabilities of the C++ half, and thus can forward invocations implemented on the C++ half into the other language. OTcl is an OBJECT SYSTEM LAYER that adds dynamic object-oriented language concepts to the static languages C and C++.

XOTcl [NZ00] is the successor of the language OTcl. We have implemented a C++/XOTcl binding using the wrapper generator SWIG [Swi03]. This project has similar goals and a similar architecture as the TclCl solution explained above. The main difference is that in this solution the variable/method bindings are not registered in the C++ constructors, but are generated from an INTERFACE DESCRIPTION (the SWIG header file – which is similar to a C++ header file).

As Tcl extensions, both OTcl and XOTcl use strings in scripts to represent primitive data types. Internally, all primitive data types of XOTcl, such as ints, floats, doubles, booleans, strings, lists, and XOTcl objects are presented by so-called `Tcl_Objs`. These are AUTOMATIC TYPE CONVERTERS which contain data structures for a string representation and an internal representation for the respective type implemented in C. Also they contain conversion routines for each type to strings and for strings to each type – also implemented in C. `Tcl_Objs` use an abstract interface – using C function pointers – so that the Tcl INTERPRETER can automatically convert each type without knowing any more details of the types.

Configuring Aspects using Split Objects

A more sophisticated SPLIT OBJECT solution [Zdu04] is implemented in Frag [Zdu03a]. Frag is a Tcl variant that can be completely embedded in Java. It is implemented on top of Jacl [DR03], a Tcl interpreter implemented in Java. Frag also exploits SPLIT OBJECTS for language integration, but in contrast to the OTcl and XOTcl solutions explained above, it does not use registration or generative programming for access of the other split object half, but Java Reflection. Frag thus can be embedded in Java as a COMMAND LANGUAGE and be dynamically connected to Java classes.

Within Frag, there is a Frag class `JavaClass` provided. This class is used for wrapping a Java class with a Frag object. When an object is derived from this class or any of its subclasses, a SPLIT OBJECT consisting of a Java half and a Frag half is created.

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. Both, wrapping a Java class and forwarding invocations, is implemented using Java reflection. Internally, primitive Java types are automatically converted to and from strings. Non-primitive Java types have to be used as SPLIT OBJECTS in order to be accessed from Frag.

When the goal is to configure a Java application, typically the Frag script are used from within Java. However, Java invocations require the developer to know which objects are SPLIT OBJECTS. Consider a Java object `circle` is defined as a SPLIT OBJECT and the following Java invocation is performed:

```
circle.setRadius(2.0);
```

This Java invocation bypasses the COMMAND LANGUAGE Frag. What would be needed instead is an invocation sent through the COMMAND LANGUAGE:

```
frag.eval("circle setRadius 2.0");
```

The pattern HOOK INJECTOR [Zdu03b] provides a solution. For different languages different tools exist that can perform static hook injections. For instance, for Java we can use AspectJ. The following aspect associates statically with a Frag INTERPRETER. It contains one advice that is invoked before the constructors of user-defined classes. It calls `makeSplitObject` to create 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;
  ...
  protected static void makeSplitObject(
    JoinPoint jp, Object o) {...}
  protected static Object invokeSplitObject(
    JoinPoint jp, Object o) {...}

  abstract pointcut splitObjectClasses(Object obj);
  pointcut theConstructors(Object obj):
    splitObjectClasses(obj) && execution(new(..));
  pointcut theMethods(Object obj):
    splitObjectClasses(obj) &&
    execution(* *(..)) &&
    !execution(String toString());

  before(Object obj): theConstructors(obj) {
     makeSplitObject(thisJoinPoint, obj);
  }
  Object around(Object obj) : theMethods(obj) &&
    !cflow(execution(* invokeSplitObject(..))) {
      return invokeSplitObject(thisJoinPoint, obj);
  }
}
```

Note that this aspect excerpt is simplified. The actual aspect implementation in the Frag distribution is more complex because it treats all Java primitive types by separate pointcuts and advices. (This terminology stems from AspectJ and is explained in detail in [KHH$^+$01]).

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

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

The default behavior of this aspect is that all invocations are sent through the COMMAND LANGUAGE, but the invocations are not altered. By re-defining the classes in the dynamic Frag language, one can dynamically compose the internals of the aspect. Thus, here we have applied the SPLIT OBJECT pattern for integrating AspectJ aspect with the Frag COMMAND LANGUAGE. That is, we can compose and configure aspect implementations dynamically at runtime. The same Frag code used for specifying a Java aspect can also be used to specify an aspect in another language (e.g. Tcl, C, or C++). We only have to use another aspect framework for weaving the aspect.

Conclusion

In this paper, we have presented a number of patterns for software architecture integration from the specific areas of language integration and component integration. The patterns play a central role in a huge number of integration architectures. There are many external patterns and many patterns missing in this broad area yet in order to provide a full pattern language for this important field.

Acknowledgments

Many thanks to Charles Weir for his attention as a EuroPLoP 2004 shepherd and his constructive comments. Also, thanks to the participants of the EuroPLoP 2004 writers' workshop who provided valuable feedback.

References

[BCK98] L. Bass, P. Clement, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, USA, 1998.

[Bec04] Dave Beckett. Redland RDF application framework. http://www.redland.opensource.ac.uk/, 2004.

[BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP/, 2000.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-orinented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.

[DR03] M. DeJong and S. Redman. Tcl Java Integration. http://www.tcl.tk/software/java/, 2003.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GNZ00] M. Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceedings of EuroPlop 2000*, Irsee, Germany, July 2000.

[GNZ01] M. Goedicke, G. Neumann, and U. Zdun. Message redirector. In *Proceedings of EuroPlop 2001*, Irsee, Germany, July 2001.

[GZ02] 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.

[JW98] R. Johnson and B. Woolf. Type object. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, October 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, Finnland, June 1997. LCNS 1241, Springer-Verlag.

[NZ00] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.

[OGJ03] J. Oberleitner, T. Gschwind, and M. Jazayeri. Vienna component framework enabling composition across component models. In *Proceedings of the International Conference on Software Engineering (ICSE 2003)*, Portland, Oregon, USA, May 2003.

[Ope00] Open Mash Consortium. The open mash consortium. http://www.openmash.org, 2000.

[PSDF01] 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*, Kyoto, Japan, Sep 2001.

[SSRB00] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.

[Swi03] Swig Project. Simplified wrapper and interface generator. http://www.swig.org/, 2003.

[UCB00a] UCB Multicast Network Research Group. Network simulator - ns (version 2). http://www.isi.edu/nsnam/ns/, 2000.

[UCB00b] UCB Multicast Network Research Group. Tclcl. http://www.isi.edu/nsnam/tclcl/, 2000.

[VKZ04] M. Voelter, M. Kircher, and U. Zdun. Remoting Patterns, 2004. to be published in Wiley's pattern series.

[VZ02] O. Vogel and U. Zdun. Content conversion and generation on the web: A pattern language. In *Proceedings of EuroPlop 2002*, Irsee, Germany, July 2002.

[WL95] D. Wetherall and C. J. Lindblad. Extending Tcl for dynamic object-oriented programming. In *Proc. of the Tcl/Tk Workshop '95*, Toronto, July 1995.

[Zdu03a] U. Zdun. Frag. http://frag.sourceforge.net/, 2003.

[Zdu03b] U. Zdun. Patterns of tracing software structures and dependencies. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.

[Zdu04] U. Zdun. Using split objects for maintenance and reengineering tasks. In *8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, Tampere, Finland, Mar 2004.