Patterns of Argument Passing

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

Argument passing means passing values along with an invocation. Most programming languages provide positional arguments as their ordinary argument passing mechanism. Sometimes ordinary argument passing is not enough, for instance, because the number of arguments or their types differ from invocation to invocation, or optional arguments are needed, or the same arguments are passed through a chain of multiple receivers and must vary flexibly. These issues can be resolved using ordinary argument passing mechanisms, but the solutions are usually cumbersome. In many systems, such as programming languages, programming environments, frameworks, and middleware systems, advanced argument passing solutions are provided to better address these issues. In this paper we present four patterns applied in these advanced argument passing solutions: VARIABLE ARGUMENT LISTS allow an operation to receive arbitrary numbers of arguments, OPTIONAL ARGUMENTS let operations have arguments which can either be provided in an invocation or not, NON-POSITIONAL ARGUMENTS allow arguments to be passed in any order as name/value pairs, and CONTEXT OBJECTS are special types used for the purpose of argument passing.

Introduction

Argument passing

Argument passing is an integral part of all forms of invocations, for instance, performed in object-oriented and procedural systems. All systems that provide facilities for performing invocations thus must provide some way to pass arguments (also called parameters) to operations. As a first example for such systems one might think of programming language implementations, such as interpreters, compilers, and virtual machines. But argument passing is also relevant for any other kind of system that performs invocations on top of the facilities offered by a programming language, such as middleware systems, aspect-oriented composition frameworks, component frameworks, interactive shells of operating systems or programming languages, enterprise integration frameworks, and so forth.

Intended audience

In this paper we present some patterns that provide advanced argument passing solutions. These patterns are important for developers of the systems named above, when they want to provide some argument passing mechanism in a language or framework. In addition to that, the patterns are also relevant for developers using these systems, because in some situations the ordinary (i.e. positional) argument passing mechanisms offered by the language or framework do not cope well with a particular design problem. Then it is advisable to write a little argument passing framework on top of the language or framework that supports the respective pattern.

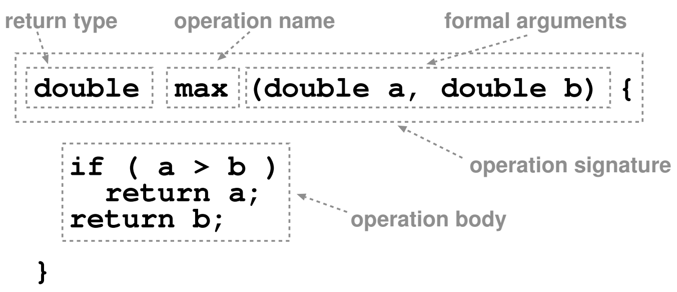

Figure 1: Operation definition

Terminology To explain the patterns, we use the following terminology applicable to all kinds of languages and frameworks mentioned above: any kind of function, procedure, method, etc., be it local or remote, is called an *operation*. Each operation has an *operation signature*. The signature contains at least an *operation name* and a list of *arguments*. In typed environments the signature also contains types for each argument and a return type. An operation is “called” or “invoked” using an *invocation*.

In the text below, we sometimes refer to “ordinary arguments”. With this term we mean the typical, positional arguments offered by almost any procedural or object-oriented programming language, such as C, C++, Java, Tcl, etc.

The arguments in the signature are called *formal arguments* because they act as placeholders for argument values provided by the invocation. The concrete argument values provided by an invocation are called *actual arguments*. Each *actual argument* is mapped to the *formal argument* at the same position.

When the invocation takes place, each formal argument is filled with the value of the respective actual argument. This can be done by copying the value of the actual argument into the storage space of the operation (call-by-value), as opposed to providing only the address of the storage space of the actual argument to the operation (call-by-reference). Another scheme of argument passing is call-by-name, which refers to passing the (unevaluated) code of the argument expression to the operation, and this code is evaluated each time the argument is accessed in the operation. In dynamic languages call-by-name can be applied at runtime (examples are arguments evaluated using `eval` in Lisp or Tcl), or it can be performed statically by compilers or preprocessors (e.g. C macros). There are a number of other, less popular parameter passing schemes, such as call-by-value-return in Fortran, or copy-a-value-in and copy-it-back.

The terms are illustrated using a Java method and invocation as an example in Figures 1 and 2.

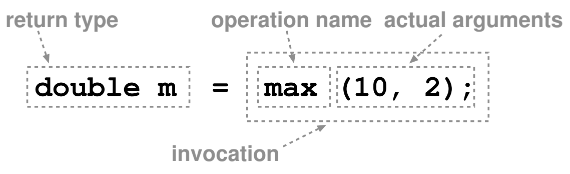

Figure 2: Operation invocation

Pattern Language Outline

The following patterns are presented in this paper:

- A VARIABLE ARGUMENT LIST provides an argument passing mechanism with a special syntax, which allows the client to invoke the operation using any number of arguments for this last argument. The actual arguments are put as a list into the last formal argument.
- OPTIONAL ARGUMENTS are an argument passing mechanism that uses a special syntax to denote that one of the formal arguments is optional. A default value is provided, which is used in case the client does not provide the OPTIONAL ARGUMENT in an invocation.
- NON-POSITIONAL ARGUMENTS are an argument passing mechanism that allows clients to provide arguments in an invocation as name/value pairs. Because each argument is named, the arguments can be provided in any order, and the argument passing mechanism automatically matches the correct actual arguments to the formal arguments.
- A CONTEXT OBJECT is a special object type that is used for argument passing. This object type is used as an argument of the operation (often it is the only argument), and the arguments to be passed to the operation are encapsulated in the CONTEXT OBJECT (e.g. as instance variables).

Related Patterns

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

- An AUTOMATIC TYPE CONVERTER [Zdu04b] converts types at runtime from one type to another. There are two main variants of the pattern: one-to-one converters between all supported types and converters utilizing a canonical format to/from which all supported types can be converted. An AUTOMATIC TYPE CONVERTER is primarily used by the patterns presented below for realizing type conversions.
- In a REFLECTION [BMR+96] architecture all structural and behavioral aspects of a system are stored into meta-objects and separated from the application logic components. The latter can query the former in order to get information about the system structure. In an argument passing architecture, REFLECTION is especially used to introspect ordinary operation signatures to perform a mapping between the arguments passed using the patterns and ordinary invocations.
- The CONTEXT OBJECT pattern is a general pattern for passing arguments using a special object type. Various special-purpose variants of this pattern have been described in the literature before: INVOCATION CONTEXTS [VKZ04] are CONTEXT OBJECTS used in distributed invocations; MESSAGE CONTEXTS [Zdu03, Zdu04a] are CONTEXT OBJECTS used in aspect-oriented composition frameworks and interceptor architectures; ARGUMENTS OBJECT [Nob97] is an object that contains all elements of an operation signature, for instance, as variables; ANYTHING [SR98] describes a generic data container; ENCAPSULATE CONTEXTS [Kel03] are CONTEXT OBJECTS used to encapsulate common data used throughout the system; OPEN ARGUMENTS [PL03] are CONTEXT OBJECTS used to support a dynamic set of arguments.
- A PROPERTY LIST [SR98] is a data structure that allows developers to associate names with arbitrary values and objects. This structure is needed to represent a simple list of

NON-POSITIONAL ARGUMENTS. The PROPERTY LIST pattern thus can be used to internally implement the NON-POSITIONAL ARGUMENTS pattern.

Figure 3 shows an overview of the patterns described in this paper and the relationships explained above. The patterns described in this paper are rendered in black, the related patterns in grey.

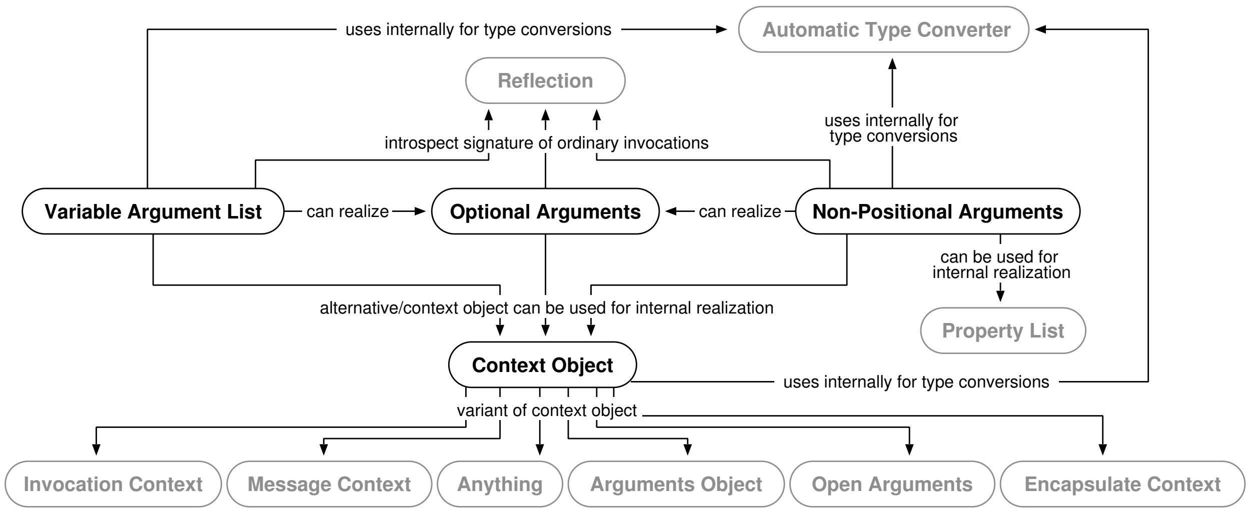
uses internally for type conversions
Automatic Type Converter
Reflection
introspect signature of ordinary invocations
uses internally for type conversions
Variable Argument List
can realize
Optional Arguments
can realize
Non-Positional Arguments
can be used for internal realization
Property List
alternative/context object can be used for internal realization
Context Object
uses internally for type conversions
variant of context object
Invocation Context
Message Context
Anything
Arguments Object
Open Arguments
Encapsulate Context

Figure 3: Pattern map: Most important relationships of the patterns

Variable Argument List

Context

You are writing operations to be invoked for instance in a programming language or in a distributed system.

Problem

A particular operation needs to receive a varying number of arguments, and you do not know in advance how many arguments will be received. You only know that the arguments to be received are all of the same type, and they can be treated in a uniform way. Ordinary operation signatures, however, cannot retrieve arbitrary numbers of arguments. Thus you have to apply tedious and error-prone workarounds for this situation, such as abusing polymorphism (e.g. overloading) or passing the arguments in a helper data structure.

Forces

Consider you want to process a list of objects, but do not know in advance how many arguments are in the list. For small numbers of objects, you can use overloading to be able to invoke the operation with a varying number of arguments:

```
void processList(Object o1) {
  //...
}
void processList(Object o1, Object o2) {
  //...
}
void processList(Object o1, Object o2, Object o2) {
  //...
}
```

Besides the problem that you have to write a huge number of unnecessary operations, you face the problem that this approach does not scale well for possibly larger numbers of arguments: consider you might receive lists with up to 1000 arguments. You would have to write 999 unnecessary operations.

An alternative solution for this problem is to bundle the arguments in a collection data structure (such as a list or an array). But this solution is quite complex because in each such operation you have to process the arguments in the list, and for each invocation you have to fill the data structure before you can perform the invocation.

A collection data structure requires the caller to put the appropriate arguments into the data structure, which make the caller more complex. Note that the overloading solution sketched above, in contrast, makes the callee more complex and error prone.

Another problem of using a collection data structure is that two kinds of invocations exist in the system. Rather it would desirable that all invocations look the same.

Solution

Provide a special syntax for the VARIABLE ARGUMENT LISTS that might be added as the last argument, or the only argument, to the argument lists of operations. Each argument in the VARIABLE ARGUMENT LIST is of the same type, which might be a generic type such as string or void pointer. The language implementation (e.g. the compiler or interpreter) or the framework implementation (e.g. the distributed object framework) provides a

functionality to process the VARIABLE ARGUMENT LISTS. Thus, from the developers perspective, all invocations of VARIABLE ARGUMENT LIST operations look just like ordinary invocations, except that they vary in their length. Also, provide an API to make the arguments passed through a VARIABLE ARGUMENT LIST accessible from within the operation.

Discussion The arguments of VARIABLE ARGUMENT LIST must be distinguishable in the invocation from other arguments. That's the reason why VARIABLE ARGUMENT LIST are usually realized as the last argument in the operation. An alternative is to delimit them in the invocation, for instance using a special character. But this would violate the goal that invocations with VARIABLE ARGUMENT LIST should look the same as ordinary invocations.

In principle it is also possible to have VARIABLE ARGUMENT LISTS be placed in the middle of ordinary arguments. This, however, is not advisable because in this case it is easy that bugs, such as wrong number of arguments, are not detected.

Similarly, a simple, working solution is to allow only for one VARIABLE ARGUMENT LIST per operation. In principle it is also possible to have more VARIABLE ARGUMENT LISTS in one operation, if the arguments can be distinguished by their type. Again, this might lead to bugs that are hard to find, for instance, when one of the argument types can be automatically converted to another one.

In type-safe environments, type-safety is an issue when using VARIABLE ARGUMENT LISTS. The typical solution is to let all arguments in the VARIABLE ARGUMENT LIST be of the same type. Otherwise, it would be necessary to define how to handle the different types and maybe delimit them, meaning that VARIABLE ARGUMENT LISTS would have a pretty different appearance in the signature than ordinary arguments (which is usually not wanted). If different types are needed in a VARIABLE ARGUMENT LIST, a super-type of these types can be used for defining the VARIABLE ARGUMENT LIST or, if this is not possible, a generic type, such as string, `void*`, or `Object`. A VARIABLE ARGUMENT LIST in an untyped environment is equivalent to using a generic type in typed environments.

In summary, in most cases it is advisable to allow for only one VARIABLE ARGUMENT LIST per operation signature and enforce that this VARIABLE ARGUMENT LIST is the last argument of the operation signature. All arguments of the VARIABLE ARGUMENT LIST are passed as the same type.

Note that we require a way to retrieve the arguments in the VARIABLE ARGUMENT LIST from within the operation. Here, VARIABLE ARGUMENT LIST arguments must be a bit different than ordinary arguments, because in the one operation signature element that represents the VARIABLE ARGUMENT LIST *n* arguments are hidden. Usually, an API or special syntax is provided, which provides a way to (a) retrieve the list of variable arguments (e.g. as a list data structure) and (b) find out how many variable arguments are passed through. Using this information, the VARIABLE ARGUMENT LIST can be processed using the operations of the list data structure.

Consequences VARIABLE ARGUMENT LISTS solve a prevalent concern in writing generic and reusable operations. They are an elegant solution because they are applied automatically and do not look much different to ordinary invocations. Only the operation implementation must be written in a slightly different style.

If VARIABLE ARGUMENT LIST are not language-supported or framework-supported, some effort to provide an implementation is required. A simple emulation (e.g. using a collection data structure) is not much work, but one also has to write a little program generator to convert the invocation to the VARIABLE ARGUMENT LIST format.

A much simpler, but slower solution is to use strings (or other generic types) for argument passing and an AUTOMATIC TYPE CONVERTER to convert the invocations back and forth. An invocation:

```
processList(3, 1, 2, 3);
```

would then become:

```
processList("3, 1, 2, 3");
```

This is not very desirable in the context of many programming language because again we would end up with two different styles of invocations. Moreover, the solution is rather slow because back and forth conversion to strings is required. But there are situations were this solution is highly applicable. For instance, in string-based programming languages (such as most scripting languages) there is no difference in the invocation styles. Or, in middleware implementations the invocations are sent as a byte-array over the wire anyway. Thus, again, there is no difference to all other invocations.

VARIABLE ARGUMENT LISTS can make overloading resolution more complex, ambiguous, or, in some situations, even impossible. Thus usually it is advisable not to use overloading for an argument that is realized as a VARIABLE ARGUMENT LIST, or at least introduce an unambiguous rule for overloading VARIABLE ARGUMENT LISTS.

Type-safety might be compromised, depending on the VARIABLE ARGUMENT LISTS implementation (compare the C/C++ and Java known uses below).

Known Uses Some known uses of the pattern are:

- In C and C++ VARIABLE ARGUMENT LISTS are language-supported. In place of the last argument you should place an ellipsis ("`...`"). C and C++ provide an API to process the VARIABLE ARGUMENT LIST (starting with `va_`) as in the following example:

  ```
  void processList(int n, ...) {
    va_list ap;
    va_start(ap, n);
    printf("count = %d: ", n);
    while (n-- > 0) {
      int i = va_arg(ap, int);
      printf("%d ", i);
    }
    printf("\n");
    va_end(ap);
  }
  ```

  This operation can be used like any other operation:

```
processList(1, 1);
processList(3, 1, 2, 3);
```

Please note that functions that take a variable number of arguments ("varargs") are generally discouraged in C/C++ style guides (see e.g. [CEK+00]) because there is no truly portable way to do varargs in C/C++. If varargs are needed, it is advisable to use the library macros for declaring functions with VARIANT ARGUMENT LISTS.

- In the scripting language Tcl (similar to other scripting languages) a special argument `args` can be provided to an operation as the last argument. In this case, all of the actual arguments starting at the one that would be assigned to `args` are combined into a list. This combined value is assigned to the local variable `args`, which is an ordinary Tcl list.

- Leela [Zdu04c] is a Web services framework that uses VARIABLE ARGUMENT LISTS for generic argument passing between Web services. A Leela service is bound to a SOAP endpoint, and this endpoint offers a string-based interface. This interface is mapped to the Web service operation using REFLECTION (see also the pattern INTROSPECTION OPTIONS [Zdu03]).

- In Java, starting with version 5.0, Var-Args are provided. Java's solution is similar to the C solution. A major difference is that it is type-safe. For instance, we can specify an operation for processing a `String` list:

  ```
  public static void processList(String... args) {
    for (String a : args) {
      System.out.println(a + " ");
    }
  }
  ```

  Java's Var-Args can receive any argument type by using a more generic type, such as Java's `Object`, for instance.

- Many programming languages provide a VARIABLE ARGUMENT LIST mechanism to receive arguments from the command line. This design is due to the argument passing interface of command shells, especially UNIX shells, which led to C/C++'s "`int main(int argc, char *argv[])`" interface to programs. Most contemporary programming languages support a similar interface, for instance, in Java, command line arguments are mapped to a special `String` array that is the argument of the operation "`static void main(String[] args)`".

Optional Arguments

Context You are writing operations to be invoked for instance in a programming language or in a distributed system.

Problem **Sometimes one operation can be defined for a varying number of arguments. This situation can in principle be solved using VARIABLE ARGUMENT LISTS. But consider the situation is slightly different to the problem solved by VARIABLE ARGUMENT LISTS: you know the possible arguments in advance, and the number of arguments is manageable. The arguments might be of different types (or kinds in untyped languages); that is, they cannot or should not be treated uniformly.**

Forces Constructors are operations that should be able to receive differing numbers of arguments because different clients want to configure different values. All unspecified values should be filled with default values. Consider the following Java code as an example:

```
class Person {
  Name name;
  Address homeAddress;
  Address workAddress;
  ...
  Person(Name _name, Address _homeAddress, Address _workAddress) {
    name = _name;
    homeAddress = _homeAddress;
    workAddress = _workAddress;
  }
  Person(Name _name) {
    this(_name, null, null);
  }
  ...
}
```

In this example, the variables `homeAddress` and `workAddress` are optional and have `null` as a default value. To realize this concern, the `Person` constructor needs to be defined twice, just to pass the default values to the operation that really does the work. Usually, there are more such constructors, and we need to provide similar forwarders in subclasses as well. For instance, to provide the option that the work address is optional, another constructor has to be added.

The solution in the example uses Java's method overloading which works by realizing a concern using multiple operations with different signatures and possibly chaining them with invocations among each other (as in the example above). This is a heavy-weight solution for the simple problem of realizing an optional default value. For each optional argument, and each possible combination of optional arguments, we need to provide one additional operation. The result is a lot of unnecessary lines of code, reducing the readability of the program.

Another problem is that we cannot provide all possible combinations of arguments because Java's overloading mechanism selects methods only on basis of the signature of the operation.

Sometimes the types of arguments conflict, for instance, in the above example we cannot provide default values for both `homeAddress` and `workAddress`, because the two operation signatures:

```
Person(Name _name, Address _homeAddress);
```

and:

```
Person(Name _name, Address _workAddress);
```

are conflicting. The compiler cannot distinguish between them because they have the same types in their signature.

Note that it is not elegant to use VARIABLE ARGUMENT LISTS in this and similar examples. The arguments of constructors are named and typed. With a VARIABLE ARGUMENT LIST you would have to pass all the arguments using a generic type, and then obtain the individual arguments using their position in the VARIABLE ARGUMENT LIST. This approach makes it hard to handle changes in argument lists.

Solution **Introduce a special syntax for operation signatures to mark some arguments as OPTIONAL ARGUMENTS. For each OPTIONAL ARGUMENT provide a default value. Provide a language-support or framework-support for selecting or passing arguments to operations who have OPTIONAL ARGUMENTS. It is important that there are no syntactic ambiguities which actual argument belongs to which formal argument.**

Discussion OPTIONAL ARGUMENTS require default values because without them it would be undefined how to handle an invocation in which an OPTIONAL ARGUMENT is omitted. Default values can be provided in different fashions:

- They can simply be provided in the operation signature, where the optional argument is defined.
- They can be looked up at runtime and added to the actual invocation by the language implementation or framework. To use this variant is advisable if the default values should be modifiable after the program has been compiled or started. For example, the default values can be defined in a configuration file or an external repository.
- They can be defined programmatically: some code handles the situation when an OPTIONAL ARGUMENT is not provided by an invocation.
- They can be implicitly defined, for instance, by some convention. For example, if there is an "empty" value or system-wide default value, this value can be chosen by the language or framework if no value for the OPTIONAL ARGUMENT is given. If there is an old value (e.g. from previous invocations), also the old value can be used as the default value.

The OPTIONAL ARGUMENTS pattern is often combined with other patterns. A VARIABLE ARGUMENT LIST is implicitly an OPTIONAL ARGUMENT that defaults to "empty". When the OPTIONAL

ARGUMENTS pattern is combined with VARIABLE ARGUMENT LISTS, it is important that the order of the two patterns in argument lists is clearly defined, so that there are no ambiguities. NON-POSITIONAL ARGUMENTS are often OPTIONAL ARGUMENTS, meaning that an omitted NON-POSITIONAL ARGUMENT is treated as being optional. A CONTEXT OBJECT implementation might also provide support for OPTIONAL ARGUMENTS.

Consequences OPTIONAL ARGUMENTS provide a look and feel similar to ordinary invocations. They can be applied automatically. In the operation signature, a special syntax is required for defining an argument as being optional and for defining or retrieving the default value. Usually invocation and operation bodies do not have to be adapted to be used with OPTIONAL ARGUMENTS.

In compiled languages, the default values cannot be changed at runtime. For a change of a default value a recompilation is necessary.

Known Uses Some known uses of the pattern are:

- In a C++ operation definition, the trailing formal arguments can have a default value (denoted using "="). The default value is usually a constant. An example is the following operation signature, which receives two `int` arguments, the second one being optional with the default value `5`:

  ```
  void foo(int i, int j = 5);
  ```

- Many scripting languages support OPTIONAL ARGUMENT for operations. In Tcl [Ous94], for instance, OPTIONAL ARGUMENTS can be defined as pairs of argument name and default value. OPTIONAL ARGUMENTS need not be specified in an operation invocation. However, there must be enough actual arguments for all the formal arguments are not OPTIONAL ARGUMENTS, and there must not be any extra actual arguments. For instance, the following `log` procedure has an optional argument `out_channel`, which is per default configured to the standard output:

  ```
  proc log {log_msg {out_channel stdout}} {
    ...
  }
  ```

  If the last formal argument has the name `args`, it is treated as a VARIABLE ARGUMENT LIST. In this case, all of the actual arguments starting at the one that would be assigned to `args` are passed as a VARIABLE ARGUMENT LIST. That is, it is not possible that there are ambiguities between the OPTIONAL ARGUMENTS and the arguments for the VARIABLE ARGUMENT LIST.

- In the GUI toolkit TK [Ous94], constructors of widgets provide access to the widget options, such as background, width, colors, texts, etc., as OPTIONAL ARGUMENTS, which represent either empty values (like an empty text) or values that are often chosen (e.g. the color of the surrounding widget). A TK widget can therefore be initiated with only a very few lines of code because only those options that differ from the defaults must be provided. For example, the following code instantiates a button widget and configures it with the label "Hello" and a callback command that prints "Hello, World!" to the standard output:

```
button .hello -text Hello -command {puts stdout "Hello, World!"}
```

The operation `configure` allows TK programs to access the widget options. Thus `configure` is an operation with NON-POSITIONAL ARGUMENTS in which each argument is an OPTIONAL ARGUMENT and its value defaults to the current setting of the widget. This way only those options of a widget to be changed must be specified in a `configure` invocation. For example, we can configure a red background for the button widget:

```
.hello configure -background red
```

- The GNU Program Argument Syntax Conventions [GNU05] recommend guidelines for command line argument passing. To specify an argument as an OPTIONAL ARGUMENT, a so-called long option, it is written as `--name=value`. This syntax enables a long option to accept an argument that is itself optional. Many UNIX tools and configure scripts follow this convention. For example, many configure scripts offer a number of options, such as `--prefix` and `--exec-prefix` (those are used for configuring the installation path). These arguments can optionally be appended to configure invocations:

  ```
  ./configure --prefix=/usr --exec-prefix=/usr
  ```

  If the options are omitted, they have a default value, such as `/usr/local`.

Non-Positional Arguments

Alias *Named Actual Arguments*, *Named Parameters*

Context You perform invocations, for instance, in a programming language or in a distributed system.

Problem **You need to pass arguments along with an invocation. You are faced with one of the following two problems: firstly, at the time when you design the operation which receives the arguments, you do not know how many arguments need to be passed. Different invocations of the operation require a different number of arguments. Secondly, some invocations require a large number of arguments. These invocations are hard to read because one must remember the meaning of each argument in order to understand the meaning of the whole invocation. Matters become even worse when both problems occur together, i.e. there is a large number of arguments and some of them are OPTIONAL ARGUMENTS.**

Forces Consider the following invocation:

```
ship.move(12, 23, 40);
```

This very simple invocation can only be understood with the specification of the operation `move` in the back of the mind. Developers usually have to deal with a lot of such operations at the same time, and thus it is impossible to remember the meaning of all arguments of all operations. To understand a program, one has to continuously look at the operation specifications.

This example illustrates the problem of readability that many ordinary invocations might have, once a certain number of arguments is exceeded.

Another important problem is that of extensibility. Programming languages like Java offer overloading to extend operations, such as `move` in the example above. This way we can overload an operation, and provide multiple realization of the operation. For instance, we can provide one `move` implementation that receives three integers as arguments (as above), and one `move` implementation that receives an object of type `ThrustVector`. But as overloading depends on the type system, we can only define overloaded operations with a different number of arguments and/or different types of arguments. We are not able to provide a second realization of `move` that also receives three integers.

From time to time, we make little semantic mistakes when invoking such operations with ordinary operations. For instance, we might twist two arguments in an invocation. Most of the time the compiler finds such mistakes because different types are needed, or our application code complains because the values provided are not meaningful. But sometimes such mistakes stay undetected because the twisted arguments are of the same type and the provided values are meaningful. For instance, in the above example, a little mistake like:

```
ship.move(40, 12, 23);
```

might stay undetected. Such mistakes might produce hard to find bugs.

The pattern VALUE OBJECT [Hen00a, Hen00b] provides a possible solution to this problem. A VALUE OBJECT is realized by a lightweight class that has value semantics, and is typically, but not always, immutable. If we make all values of the example operation VALUE OBJECTS, we could write the invocation as follows:

```
ship.move(Left(12), Right(23), Thrust(40));
```

Together with overloading, VALUE OBJECTS provide a well defined interface for ship movements, which is typed and supports multiple combinations of arguments. Using VALUE OBJECTS, however, requires us to change the operation signature. This might not be possible for third-party code, and thus we need to write a VALUE OBJECT wrapper for each extended third-party operation. The VALUE OBJECT solution does only work well for small numbers of possible arguments, because operation overloading means writing additional operations for each possible combination of arguments. Also, types can only be used to distinguish arguments as long as they are different (consider two `Thrust` arguments, for instance).

Solution **Provide an interface to pass NON-POSITIONAL ARGUMENTS along with an invocation. Each NON-POSITIONAL ARGUMENT consists of an argument name plus an argument value. The argument name can be matched to the respective arguments of the operation. Thus it is no longer necessary to provide the arguments in a strict order, but any order is applicable. Usually NON-POSITIONAL ARGUMENTS are OPTIONAL ARGUMENTS.**

Discussion Non-positional arguments provide each argument as a name/value pair. We need some syntax to distinguish names from values. For instance, we can start each argument name with a dash "-". Then we can write the above invocation example in a form like:

```
ship.move -left 12 -right 23 -thrust 40;
```

Of course, this form does not conform to the syntax of ordinary arguments of the programming language anymore. Thus we must implement some support for dealing with NON-POSITIONAL ARGUMENT invocations:

- We can provide program generator (preprocessor) which parses the program text, finds the NON-POSITIONAL ARGUMENT invocations, and checks that they conform to the arguments required by the operation. The preprocessor substitutes the NON-POSITIONAL ARGUMENT invocations with ordinary argument invocations.
- A more simple way to realize NON-POSITIONAL ARGUMENT invocations is to use a string-based syntax. That is, all operations receiving NON-POSITIONAL ARGUMENTS receive only one string as an argument in which the NON-POSITIONAL ARGUMENTS are encoded, such as:

  ```
  ship.move("-left 12 -right 23 -thrust 40");
  ```

  This syntax is simple, but we need to parse the string, type-convert the arguments (using an AUTOMATIC TYPE CONVERTER), and map them to the ordinary arguments. Runtime string parsing is slower than invocations injected by a program generator.

- We can provide a special kind of CONTEXT OBJECT which holds NON-POSITIONAL ARGUMENTS. That is, the CONTEXT OBJECT must be able to store a dynamic length table or list of name/value pairs, and the values must be of a generic type. Thus type conversion might get more simple than in the string-based variant, and the solution is more efficient than string parsing. The CONTEXT OBJECT, however, requires a different syntax than ordinary invocations. Thus in most cases CONTEXT OBJECTS should rather be used internally to implement NON-POSITIONAL ARGUMENTS and stay hidden from the developer.

- Finally, it is also possible that a programming language provides support for NON-POSITIONAL ARGUMENTS. Alternatively, some programming languages can be extended with support for NON-POSITIONAL ARGUMENTS. All other variants, described before, required a NON-POSITIONAL ARGUMENT framework – on top of a positional arguments implementation – for supporting the pattern.

When NON-POSITIONAL ARGUMENTS are implemented on top of positional arguments, we need some converter that is invoked between the invocation and the execution of the operation. The converter must transform the NON-POSITIONAL ARGUMENTS into positional arguments. That is, it needs to map the named actual arguments to names of the formal, positional arguments. To do so, the converter must know about the name and type of each positional argument, so that it can map the NON-POSITIONAL ARGUMENTS in the correct way. This knowledge can either be provided to the converter (e.g. at compile time or load time), or REFLECTION can be used by the converter to acquire the information at runtime.

The converter is also responsible for applying type conversions if they are necessary (e.g. using the AUTOMATIC TYPE CONVERTER pattern), and must raise exceptions in case of type violations. Note that the converter must also check for overloaded operations and other kinds of operation polymorphism, if supported by the programming language, and decide on basis of the provided NON-POSITIONAL ARGUMENTS which operation implementation needs to be invoked.

In the "programming language support" variant, the language implementation (compiler, interpreter, virtual machine, etc.) realizes the converter. In the "program generator/preprocessor" variant, the generator generates the conversion code. In the other variants, "string-based syntax" and "CONTEXT OBJECT", the developer might have to manually trigger the converter. For instance, the first lines in the invoked operation might query the arguments, or the invoking code must trigger conversion, such as:

```
system.invokeWithNonPosArgs("ship.move -left 12 -right 23 -thrust 40");
```

The converter internally needs to hold and perhaps pass around the name-value pairs. The pattern PROPERTY LIST [SR98] provides a data structure that allows names to be associated with arbitrary other values or objects. It is thus ideally suited as an implementation technique to internally represent the NON-POSITIONAL ARGUMENTS before they are mapped to the invocation. A hash table data structure is an (efficient) means to implement the PROPERTY LIST data structure (this solution is used by many scripting languages such as Perl or Tcl).

The pattern ANYTHING [SR98] is an alternative for PROPERTY LIST, where PROPERTY LIST is not sufficient. It is a generic data container for one (primitive) value of any kind or an associative array of these values. The pattern thus can also be used to implement NON-POSITIONAL ARGUMENTS. Finally, the CONTEXT OBJECT pattern can be used (only internally) to hold and pass around the NON-POSITIONAL ARGUMENTS.

All NON-POSITIONAL ARGUMENTS for which we can assume a default value are usually OPTIONAL ARGUMENTS. For instance, in the example we might want to move the ship without changing the course, or just change the course, or just change the course in one direction. Using NON-POSITIONAL ARGUMENTS with OPTIONAL ARGUMENTS we can assume the old value as default for all values not specified and then do invocations like:

```
ship.move -thrust 30;
...
ship.move -left 15 -right 23;
...
ship.move -thrust 10 -right 15;
```

Consequences The biggest advantage of NON-POSITIONAL ARGUMENTS is that they enhance readability and understandability of long argument lists. They can also be used on top of positional arguments, meaning that they can be used to enhance the documentation of invocations in a framework, without having to change the positional signature of a (given) target operation (see the SOAP example below for an example of distributed invocations).

Extensibility is also enhanced because overloading extensions of an operation can be based on the selection with an identifier (the argument name) and not only using the type of the argument. The combination with the OPTIONAL ARGUMENT pattern supports the extensibility of NON-POSITIONAL ARGUMENTS and enhances the changeability, when extensions are introduced: developers can define default values for extensions to a given operation. That is, invocations using the old version without the extension are still valid and do not need to be changed.

NON-POSITIONAL ARGUMENTS reduce the risk of mistakes during argument passing because the developer has to name the argument to which a value belongs.

A drawback of NON-POSITIONAL ARGUMENTS is that they are more verbose than positional arguments. That is, a program with NON-POSITIONAL ARGUMENTS has more lines of code. This drawback does not necessarily occur, when OPTIONAL ARGUMENTS are used together with NON-POSITIONAL ARGUMENTS and default values can be used. Consider an operation with 20 options, and you want to change only one of them. NON-POSITIONAL ARGUMENTS with OPTIONAL ARGUMENTS allow you to specify only the name and the value of that one argument: the result is an invocation with two extra words for arguments. An operation with positional arguments that changes all 20 options would require 18 words more than that (plus invocations to query the old values of the arguments not to be changed).

If the system or language does not yet support NON-POSITIONAL ARGUMENTS and you want to introduce them, depending on your solution, different changes to the system need to be made. For instance, you might have to introduce a converter, and the converter introduces a slight performance decrease. In some solutions, discussed above, the signature or implementation of the operations must be changed. Other solutions (like the language-based or generator-based variants) require more efforts for implementing them. The efforts and drawbacks of the individual solutions need to compared to the benefits of NON-POSITIONAL ARGUMENTS.

If positional arguments are supported as well, two styles of invocation are present. The syntax for both variants should be distinctive, so that developers can see at first glance, which kind of invocation is used or required by an operation.

Known Uses Some known uses of the pattern are:

- The SOAP protocol [BEK+00], used for Web services communication, uses NON-POSITIONAL ARGUMENTS. For instance, an invocation of an operation `GetPrice` with one argument `Item` might look as follows:

  ```
  <soap:Body>
     <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
        <m:Item>Apples</m:Item>
     </m:GetPrice>
  </soap:Body>
  ```

  In SOAP the response message also contains NON-POSITIONAL ARGUMENTS.

  Web services frameworks implemented in languages that do not support NON-POSITIONAL ARGUMENTS must map SOAP's NON-POSITIONAL ARGUMENTS to the positional arguments of the programming language. For instance, the Web services framework Apache Axis [Apa04] contains an AUTOMATIC TYPE CONVERTER which maps the NON-POSITIONAL ARGUMENTS delivered in SOAP messages to Java invocations, and vice versa.

- The GUI toolkit TK provides NON-POSITIONAL ARGUMENTS for configuring the TK widgets. Each widget has a huge number of options. Most of the time it is enough for a Widget instance to configure only a few of these options. For both readability and extensibility reasons, it is not a good choice to perform the configuration of the widget options using operations and operation overloading, as used by many other GUI toolkits. For instance, a button widget with NON-POSITIONAL ARGUMENTS can be created like this:

  ```
  button .b -text Hello! -command {puts hello}
  ```

  We can also send any of the possible widget arguments as NON-POSITIONAL ARGUMENTS to the widget for reconfiguration. For instance, we can change the font like this:

  ```
  .b configure -font {Times 16}
  ```

  The advantage of NON-POSITIONAL ARGUMENTS are that we can choose any of the 32 widget options in TK 8.4 for a button in any order and that we can directly see which option is configured in which way. TK constantly evolves. For instance, in TK 8.0 the button widget had only 28 options. Nevertheless TK 8.0 scripts usually work without changes, compatibility operations, or other measures, because the NON-POSITIONAL ARGUMENTS are combined with OPTIONAL ARGUMENTS.

- OpenACS [GA99] is a toolkit for building scalable, community-oriented Web applications on top of a Web server. It uses the Tcl scripting language as a means for developers to add user-defined operations and call them from Web pages (or Web page templates). One means to support flexible operations are so-called `ad_proc` operations. These operations can be declared with regular positional arguments, or with NON-POSITIONAL ARGUMENTS. In addition, when NON-POSITIONAL ARGUMENTS are used, it is possible to specify which ones are required, optional, and boolean. Optional arguments require a default value. They are an implementation of the OPTIONAL ARGUMENTS pattern. An example is:

```
ad_proc -public auth::authenticate {
  {-username:required}
  {-domain ""}
  {-password:required}
} {...} {...}
```

In this operation signature, the arguments `username` and `password` are required, the `domain` argument is an OPTIONAL ARGUMENT, which defaults to an empty string.

- XOTcl [NZ00] is an object-oriented Tcl variant which supports NON-POSITIONAL ARGUMENTS for all its operations. Its model is similar to that of the OpenACS framework.

Context Object

Context You are invoking operations, for instance, in a programming language or distributed object system.

Problem **You want to deliver complex or varying information to an operation. For instance, there is a huge number of arguments, the number of arguments varies from invocation to invocation, or there are OPTIONAL ARGUMENTS. So ordinary, positional arguments are not really working well here. In addition to passing the information to the operation, you need to process the information in some way. For instance, you might have to transform them into a different format (e.g. marshal them to transport them over a network). Or the same information is passed through multiple operations and each one can add or remove some information. The arguments might be of different types (or kinds) and thus cannot be treated uniformly. So the patterns VARIABLE ARGUMENT LIST or NON-POSITIONAL ARGUMENTS do not resolve all concerns either.**

Forces Consider information that is passed through multiple operations, and each operation can add or remove arbitrary information. For instance, this situation is typical for realizations of the patterns CHAIN OF RESPONSIBILITY [GHJV94], INVOCATION INTERCEPTOR [VKZ04], and PIPES AND FILTERS [BMR$^+$96, SG96]. Using ordinary, positional arguments is cumbersome here because each operation would have to know the signature of its successor to be able to invoke it. Thus the modifiability of this architecture would be limited: the operations could not be assembled in arbitrary order.

A VARIABLE ARGUMENT LIST could help to avoid this problem because all operations would just receive the VARIABLE ARGUMENT LIST and thus have the same signature. The operations could be assembled in any order. But, as a drawback, each operation would have to process the VARIABLE ARGUMENT LIST before the arguments could be accessed or changed. This means a slight performance overhead. Also, it should be possible to reuse the code for processing the list in different operations because likely most of them will process the list in more or less the same way – which is not supported directly by the VARIABLE ARGUMENT LIST pattern. VARIABLE ARGUMENT LIST only support arguments of the same kind. If there are different types, for instance, conversion to and from a generic format would be necessary.

If just a variable number of named arguments is needed, NON-POSITIONAL ARGUMENTS might resolve the problem. If the arguments or the processing requirements are more complex, however, this won't work well either, because NON-POSITIONAL ARGUMENTS do not support complex processing instructions.

Solution **Pass the arguments in a special object type, a CONTEXT OBJECT. This object provides all arguments as instance variables. The class of the object defines the structure of the CONTEXT OBJECT and the operations required to process the arguments. Using ordinary inheritance, more special CONTEXT OBJECTS can be derived.**

Discussion A CONTEXT OBJECT must be instantiated and filled with values (e.g. with the actual arguments to be passed to an operation). A typical example looks as follows:

```
Context c = new Context();
```

```
c.setValue("left", new Integer(12));
c.setValue("right", new Integer(23));
c.setValue("thrust", new Integer(40));
o1.invokeOperation(c);
```

In the operation receiving the invocation the arguments must be accessed via the CONTEXT OBJECT's API. For instance, an access to a value might look as follows:

```
Integer left = (Integer) c.getValue("left");
```

The API in this example uses key/value-pairs. This, however, is just an example, CONTEXT OBJECTS can use any kind of data structure. For instance, the same example could be realized using a special ship CONTEXT OBJECT that receives the values as instance variables, such as:

```
ShipContext sc = new ShipContext();
sc.left = 12;
sc.right = 23;
sc.thrust = 40;
o1.invokeOperation(sc);
```

A CONTEXT OBJECT is an alternative to the patterns VARIABLE ARGUMENT LIST, OPTIONAL ARGUMENTS, and NON-POSITIONAL ARGUMENTS. CONTEXT OBJECT is more generic than these patterns. This is because each of the other pattern's solutions can be realized using a CONTEXT OBJECT. However, in the concrete solutions applied by these patterns, the patterns provide more support than a solution using a generic CONTEXT OBJECT. In particular, the patterns usually allow for invocations and argument access that looks no different to ordinary invocations and argument access.

There are some style guides that advise the use of CONTEXT OBJECTS. For instance, an "old" C programming guide says: "if a function takes more than four parameters, pack them together in a struct and pass the struct instead". CONTEXT OBJECTS can be seen as the object-oriented successor of this guideline. In general, however, CONTEXT OBJECTS are especially used in infrastructure software. That is, they are often not visible to the developer, but only used internally. Examples are:

- Implementations of the patterns VARIABLE ARGUMENT LIST, OPTIONAL ARGUMENTS, and NON-POSITIONAL ARGUMENTS in interpreters, compilers, or program generators can pass the arguments within their implementation using CONTEXT OBJECTS.
- Distributed object systems need to be pass the distributed invocation through the layers of the distributed object system using a CONTEXT OBJECT. At the client side, an invocation gets step-by-step transformed into a byte-array to be sent over the wire. At the server side, the invocation of the remote object is created step-by-step from the incoming message. Again the invocation needs to be passed through multiple entities. Besides invocation information, extra context information must be transmitted, such as security information (like passwords) or transaction contexts. This special variant of the CONTEXT OBJECT pattern for distributed object frameworks is called INVOCATION CONTEXT and is described in the book *Remoting Patterns* [VKZ04].

- Aspect-oriented software composition framework need to intercept invocations and indirect them to aspect implementations. This is most often done with CONTEXT OBJECTS (see [Zdu04a]). The pattern language in [Zdu03] describes a pattern for such MESSAGE CONTEXTS[1]. The pattern is a special variant of CONTEXT OBJECTS.

In all three examples the CONTEXT OBJECTS are hidden from developers and are used inside of a framework used by the developer. An AUTOMATIC TYPE CONVERTER is usually applied by the CONTEXT OBJECT implementation to transparently convert generic types, used for instance in a distributed message, to the specific types defined by the user operation, so that the use does not have to care for type conversion.

Besides the two variants of CONTEXT OBJECTS mentioned above, INVOCATION CONTEXT [VKZ04] and MESSAGE CONTEXT [Zdu03, Zdu04a], there are more CONTEXT OBJECT variants documented in the pattern literature:

- ARGUMENTS OBJECT [Nob97] is an object that contains all elements of an operation signature, for instance, as variables. The ARGUMENTS OBJECT is passed to the operations with that signature instead of the arguments. The pattern is used in object-oriented languages as well as in procedural languages (e.g. a C `struct` can be used to encapsulate argument variables). ARGUMENTS OBJECT is a very simple variant to realize a CONTEXT OBJECT. It is advisable to use this variant, if fixed operation signatures should be simplified (or unified).
- ANYTHING [SR98] is a generic data structure that can hold any predefined primitive type, as well as associative arrays of the primitive types. Using these associative arrays, complex CONTEXT OBJECTS can be built (the arrays can contain arrays). Note that the CONTEXT OBJECT implementation in the ANYTHING pattern is scattered among multiple implementation objects. It is advisable to use this variant, if a generic data container that can be packed with arbitrary fields and values to be passed along a call chain (e.g. as in the patterns CHAIN OF RESPONSIBILITY, INVOCATION INTERCEPTOR, and PIPES AND FILTERS) is needed.
- ENCAPSULATE CONTEXT [Kel03] is a CONTEXT OBJECT that encapsulates common data used throughout the system. This pattern is used to pass the execution context for a component or a number of components as an object. The execution context can, for instance, contain external configuration data. Thus the ENCAPSULATE CONTEXT pattern describes one particular use case of the CONTEXT OBJECT pattern. Henney presents a pattern language for realizing ENCAPSULATE CONTEXT, consisting of four patterns: ENCAPSULATED CONTEXT OBJECT, DECOUPLED CONTEXT INTERFACE, ROLE-PARTITIONED CONTEXT, and ROLE-SPECIFIC CONTEXT OBJECT (see [Hen05]). These patterns are generally useful to implement CONTEXT OBJECTS.
- OPEN ARGUMENTS [PL03] are CONTEXT OBJECTS that support a dynamic set of arguments.

CONTEXT OBJECT provides a generalization of these individual patterns.

Consequences The CONTEXT OBJECT encapsulates the arguments of an operation in an object and makes them

[1]In [Zdu03, Zdu04a] this pattern in called INVOCATION CONTEXT. To avoid confusion with the same-named pattern from the book *Remoting Patterns*, we henceforth use the pattern name MESSAGE CONTEXT.

exchangeable. If a number of operations are invoked using the same CONTEXT OBJECT, data copying can be avoided: all consecutive operations work using the same CONTEXT OBJECT and pass it on to the next operation after they have finished their work. The CONTEXT OBJECT couples the data structures (arguments) and the operations that are needed to process these arguments. CONTEXT OBJECTS are extensible using ordinary object-oriented inheritance.

The downside of using CONTEXT OBJECTS is that invocations do not look like ordinary invocations, but much more code for instantiating and filling the CONTEXT OBJECTS is needed. This reduces the readability of the code. Thus CONTEXT OBJECTS are not transparent to the developer using them. Also, the operation receiving the CONTEXT OBJECTS is different to an operation using ordinary arguments. For these reasons, CONTEXT OBJECTS are often used in infrastructure software, where they are hidden from the developer.

Note that there are some situations, where a global, well known space is a simple alternative to CONTEXT OBJECTS, which avoids passing the CONTEXT OBJECTS through the whole application (an example is the environment provided to CGI programs by a web server). This alternative can easily be abused. Likewise, a danger of using CONTEXT OBJECTS is that they can be abused for tasks that are similar to those of SINGLETONS [GHJV94]. They should only be used for modular chains of invocations. A CONTEXT OBJECT that references all elements of a system and is used like a global data structure is dangerous because it strongly couples different architectural elements, meaning that the individual architectural elements cannot be understood, loaded, or tested on their own anymore.

Known Uses Some known uses of the pattern are:

- An aspect-oriented composition framework intercepts specific events in the control flow, called joinpoints. The aspect is applied to these joinpoints. Thus the operation that applies the aspect must be informed about the details of the joinpoint. Many aspect-oriented composition framework use CONTEXT OBJECTS to convey this information. For instance, AspectJ [KHH$^+$01] realizes its joinpoints using the `JoinPoint` interface. From an aspect the current joinpoint can be accessed using the variable `thisJoinPoint`. The aspect framework automatically instantiates the `JoinPoint` instances and fills it with values. For instance, the following instruction in an AspectJ advice prints the name of the signature of the currently intercepted joinpoint:

  ```
  System.err.println(thisJoinPoint.getSignature().getName());
  ```

- In the Web services framework Apache Axis [Apa04], when a client performs an invocation or when the remote object sends a result, a CONTEXT OBJECT, called the `MessageContext`, is created before a Web services message is processed. Both on client and server side, each message gets processed by multiple handlers, which realize the different message processing tasks, such as marshaling, security handling, logging, transaction handling, sending, invoking, etc. Using the `MessageContext` different handlers can retrieve the data of the message and can potentially manipulate it.

- In CORBA the Service Context is used as a CONTEXT OBJECT which can contain any value including binary data. Service context information can be modified via CORBA's Portable Interceptors.

- In .NET `CallContexts` are used as CONTEXT OBJECT. They are used to transport information from a client to a remote object (and back) that is not part of the invocation data. Examples include security credentials, transaction information, or session IDs. The data is an associative array that contains name/value pairs:

  ```
  CallContext.setData("itemName", someData);
  ```

- CONTEXT OBJECTS are often used when objects are simulated on top of procedural APIs. The first argument is a CONTEXT OBJECT which bundles all the data about the current object. For instance, the Redland API [Bec04] simulates objects using this scheme. Each Redland class has a constructor. For instance, the class `librdf_model` can be created using the `librdf_new_model` operation. A CONTEXT OBJECT of the type `librdf_model` is returned. A pointer to this CONTEXT OBJECT type is used in all operations of the `librdf_model` type. For instance, the "add" operation looks as follows:

  ```
  int librdf_model_add (librdf_model* model, librdf_node* subject,
                        librdf_node* predicate, librdf_node* object);
  ```

Conclusion

In our earlier pattern collection *Some Patterns of Component and Language Integration* [Zdu04b] we provided the starting point for a pattern language on the topic of software integration, namely component and language integration. Argument passing is an important issue in the realm of component and language integration because the argument passing styles of two systems to be integrated must be aligned. Thus, in this paper, we have supplemented our earlier patterns for component and language integration with some additional patterns. These patterns can be used by developers to realize argument passing architectures which provide more sophisticated argument passing solutions than ordinary invocations. Of course, these patterns can be applied for other tasks than component and language integration as well. As future work, we plan to further document patterns from the component and language integration domain, and integrate them into a coherent pattern language.

Acknowledgments

Many thanks to my VikingPLoP 2005 shepherd Peter Sommerlad, who provided excellent comments which helped me to significantly improve the paper.

References

[Apa04] Apache Software Foundation. Apache Axis. http://ws.apache.org/axis/, 2004.

[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.

[CEK^{+}00] L. Cannon, R. Elliott, L. Kirchhoff, J. Miller, J. Milner, R. Mitze, R. Schan, E. Whittington, N. Spencer, H. Keppel, D. Brader, and M. Brader. Recommended c style and coding standards, 2000.

[GA99] P. Greenspun and E. Andersson. Using the ArsDigita community system. *Ars-Digita Systems Journal*, Feb 1999.

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

[GNU05] GNU. Program argument syntax conventions. http://www.gnu.org/software/libc/manual% slashhtml_node/Argument-Syntax.html, 2005.

[Hen00a] K. Henney. Patterns in Java: Patterns of value. *Java Report*, (2), February 2000.

[Hen00b] K. Henney. Patterns in Java: Value added. *Java Report*, (4), April 2000.

[Hen05] K. Henney. Context encapsulation – three stories, a language, and some sequences. In *Proceedings of EuroPlop 2005*, Irsee, Germany, July 2005.

[Kel03] A. Kelly. Encapsulate context. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.

[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.

[Nob97] J. Noble. Arguments and results. In *Proceedings of Plop 1997*, Monticello, Illinois, USA, September 1997.

[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.

[Ous94] J. K. Ousterhout. *Tcl and Tk*. Addison-Wesley, 1994.

[PL03] G. Patow and F. Lyardet. Open arguments. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Addison-Wesley, 1996.

[SR98] Peter Sommerlad and Marcel Rüedi. Do-it-yourself reflection. In *Proceedings of Third European Conference on Pattern Languages of Programming and Computing (EuroPlop 1998)*, Irsee, Germany, July 1998.

[VKZ04] M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns – Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. Wiley Series in Software Design Patterns. October 2004.

[Zdu03] U. Zdun. Patterns of tracing software structures and dependencies. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.

[Zdu04a] U. Zdun. Pattern language for the design of aspect languages and aspect composition frameworks. *IEE Proceedings Software*, 151(2):67–83, April 2004.

[Zdu04b] U. Zdun. Some patterns of component and language integration. In *Proceedings of 9th European Conference on Pattern Languages of Programs (EuroPlop 2004)*, Irsee, Germany, July 2004.

[Zdu04c] Uwe Zdun. Loosely coupled web services in remote object federations. In *Proceedings of the Fourth International Conference on Web Engineering (ICWE'04)*, Munich, Germany, July 2004.