Arguments and Results

James Noble

Computer Science, Victoria University of Wellington, New Zealand, kjx@mcs.vuw.ac.nz

If an object oriented program is a collection of communicating objects, then the objects' protocols define the languages the program speaks. Unfortunately, protocols are difficult to design in isolation, so many programs' protocols are not as well designed as they could be. This paper presents six patterns which describe how objects' protocols can be designed or redesigned, by considering the arguments the protocols accept and the results the protocols return. By using these patterns, programs and designs can be made simpler, more general, and easier to change.

Introduction

Object's *protocols*, also know as *interfaces*, are very important in object oriented design. An object's protocol is the face the object presents to other objects surrounding it. This paper presents six patterns for designing object oriented protocols (see Figure 1). The patterns focus on two aspects of protocol design — the messages objects receive, and the results objects return in response to these messages. The patterns do not attempt to describe novel techniques; rather, they identify well-established solutions for object oriented design. The patterns should be useful for introducing such solutions to novice programmers, and for more experienced programmers, should illustrate when particular techniques are applicable and their relative strengths and weaknesses. The patterns are language-independent, although we present examples in Smalltalk.

The patterns form two interrelated fragments (**Patterns about Arguments** and **Patterns about Results**) which may one day form part of a larger pattern language for object oriented design. Each of the fragments has the same structure (see Figure 2) with one general pattern, and a couple of more specific patterns which specialise the general pattern to handle specialised contexts. Figure 2 also illustrates two relationships between the patterns. One pattern can *specialise* another pattern, meaning one pattern is a more specific version of the other. Alternatively, patterns can *conflict*, meaning that the patterns are mutually exclusive, each providing a different solution to a similar problem (Noble 1998, Zimmer 1994).

Pattern	**Problem**	**Solution**
Arguments Object	How can you simplify a complex protocol that has a common argument structure?	Make an Arguments Object to encapsulate the common arguments.
Selector Object	How can you simplify a protocol where several messages differ mainly in their names?	Make a single message taking an object representing the message selector as an extra argument.
Curried Object	How can you simplify an extremely complicated protocol?	Send simpler messages to an intermediary which elaborates them within its context.
Result Object	How can you return a complex answer to a question?	Return a Result Object that is the whole answer to the question.
Future Object	How can you answer a question while you think about something else?	Return a Future Object that represents the eventual result of a concurrent computation.
Lazy Object	How can you answer a question that is easy to answer now, but that may never be asked?	Return a Lazy Object that can answer the question later, if necessary.

Figure 1: Summary of the Patterns

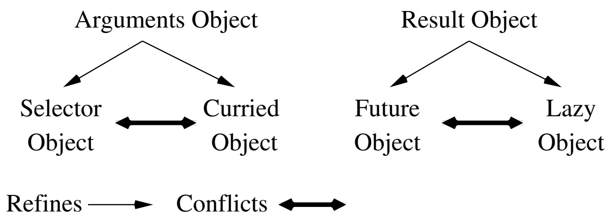

Figure 2: The Structure of the Language

Forces

Each of these patterns resolves a number of different forces. Many of the patterns consider the *ease* of *reading* or *writing* of a particular solution — generally, solutions which are easy to write are more likely to be chosen by programmers, and solutions which are easy to read are likely to be easier to maintain. Since smaller, simpler programs are generally easier to read and write, the patterns are concerned with the *complexity* or *size* of a design, such as the *number of messages* an object understands, or the *number of arguments* needed by a message. Since much complexity cannot be avoided, the patterns prefer to handle complexity once in a *server* object, rather than many times in every *client* object which uses the server.

Several patterns address the *cohesion* and *coupling* of the resulting designs, as designs with high cohesion within objects and low coupling between them are more flexible, understandable, and easier to maintain. This is often related to whether a concept is represented *explicitly* by a single object in a design, or whether it is implicit in the communication between several objects. Representing a concept explicitly as an object generally makes it easier to *identify* the concept within the design, to *change the implementation* of the concept if necessary, and to *reuse* the concept elsewhere, especially if the object represents a real concept from the program's application domain. The patterns are also concerned with *efficiency* — the *time* and *space* cost of a design, and the *number of objects* it requires at runtime.

Consequences

A common principle underlies all these patterns — that designs can often be improved by introducing (*finding*) additional objects from within the program. At first, the newfound objects seem out of place in the program, but as the program evolves, the found objects become better integrated into the design, and can be recognised as modelling concepts from the application domain.

Because they find new objects, these patterns tend to generate designs with many small, simple objects, introducing an extra level of indirection, and imposing space and time costs at runtime. All these patterns simplify a design locally (for example, by changing one particular protocol) but impose global changes to the program (because extra objects are needed to implement the changed protocol). As a result, although objects may be easier to understand in isolation, the global design of the program may become confused. This is not limited solely to these patterns. Many other patterns have this effect, including those in *Design Patterns* (Gamma, Helm, Johnson & Vlissides 1994), *Patterns of Software Architecture* (Buschmann, Meunier, Rohnert, Sommerlad & Stal 1996), and *Smalltalk Best Practice Patterns* (Beck 1997), as do Parnas's criteria for program decomposition (Parnas, Clements & Weiss 1985). To quote Alan Perlis: *In the long run every program becomes rococo — then rubble* (Perlis 1982). If these patterns are applied injudiciously they will accelerate this process.

Form

The form of the patterns is a development of the Portland form (Cunningham 1995). Each begins with a question (in italics) describing a problem, followed by one paragraph describing the pattern's context, and a second describing the forces the pattern resolves. A boldface "**Therefore:**" introduces a summary of the solution (also italicised) followed by a description of the solution in detail. Then follows an example of using the pattern, the pattern's consequences (benefits first, a boldface **But:**, then liabilities), and finally some known uses, any common variants, and any related patterns. Important forces are *italicised* when they appear in the discussion.

Patterns about Arguments

Objects communicate via protocols — a program's protocols are the glue that binds its objects together. The following three patterns describe how objects can be found within a protocol's arguments.

1 Arguments Object

How can you simplify a complex protocol that has a common argument structure?

Some objects have large and complex protocols with common internal structure. For example, a graphical View object will provide a protocol for drawing many types of objects in many different ways, but almost every message will take arguments which specify the colour, stipple, and line thickness to use when drawing. The same combinations of arguments may occur across many protocols, and as a result, many arguments may be passed in parallel through many messages and across many objects.

Large protocols are easy to use because they offer a large amount of behaviour to their clients. Unfortunately, they are often difficult or time consuming to implement, and for client programmers to learn. Every client of a large protocol depends upon the protocol's fine details, such as the names and arguments required by each message, and these dependencies make large protocols difficult to change. Moreover, large protocols are more likely to be changed than small protocols — adding an eleventh argument to a message with ten arguments is qualitatively quite different to adding a second argument to a unary message. To quote Alan Perlis: *If you have a procedure with 10 parameters, you probably missed some* (Perlis 1982).

Therefore: *Make an **Arguments Object** to encapsulate the common arguments.*

In its simplest form, an **Arguments Object** should have one variable for each argument to be eliminated from the protocol, and the usual messages to access and update its variables. Change the protocol and its implementations to accept a single **Arguments Object** in place of the eliminated arguments, and change the protocol's clients to create **Arguments Objects** as required. To support optional arguments, automatically initialise the **Arguments Object**'s variables with the arguments' default values when it is created.

Example

Consider the protocol provided by a graphical View object, as shown in Figure 3.

```
drawRectangleFrom: topLeft to: bottomRight colour: colour.
fillRectangleFrom: topLeft to: bottomRight colour: colour.
drawOvalFrom: topLeft to: bottomRight colour: colour.
```

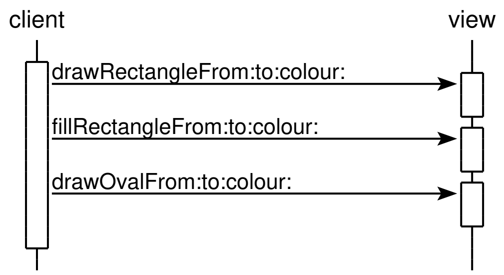

Figure 3: Basic View protocol

These messages take a number of arguments in common: topLeft, bottomRight and colour. An **Arguments Object** can eliminate these arguments from the protocol. The **Arguments Object** (which we will call a Graphic) uses three variables to replace the common arguments. Graphic's protocol includes messages to create new Graphic objects, and to read and write these variables. The View's protocol can be changed to accept a single Graphic argument, rather than the three common arguments, as shown in Figure 4.

```
"Create a Graphic"
  graphic1 := Graphic from: 10@10 to: 20@20 colour: #red.
```

```
"Draw it"
   view drawRectangle: graphic1.
"And so on..."
   graphic2 := Graphic from: 30@10 to: 40@20 colour: #blue.
   view fillRectangle: graphic2.
   graphic3 := Graphic from: 50@10 to: 60@20 colour: #green.
   view drawOval: graphic3.
```

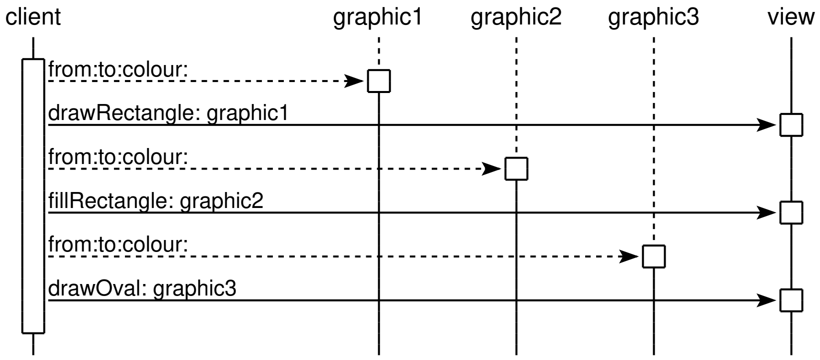

Figure 4: Arguments Object

Consequences

An **Arguments Object** makes a tradeoff between the size of a complex protocol (M messages with N common arguments) versus M messages with $N-1$ fewer arguments plus a new **Arguments Object** which requires N arguments to create it (or $1..N$ arguments if some are optional). The resulting protocols are usually easier to learn and to read. An **Arguments Object** decreases the coupling between objects involved in the protocol — objects are coupled to the **Arguments Object** object, rather than to each other — so many changes to the protocol are limited to the **Arguments Object** and its clients. Ideally, an **Arguments Object** will make an abstraction from the application domain explicit in the program's design. **But: Arguments Object** clients may be more difficult to write, as the programmer must understand both the server's protocol and the **Arguments Objects**, and create the necessary **Arguments Objects** as appropriate. While **Arguments Objects** can greatly clarify a program when they represent an application domain abstraction, using ad-hoc **Arguments Objects** indiscriminately can make a program very difficult to read, understand, and modify.

As with all the patterns in this paper, this pattern introduces an additional object into the design, requiring modifications to the program and increasing runtime space and time costs. In languages without garbage collection (such as C++) the lifetime of the **Arguments Objects** must be managed explicitly. An **Arguments Object** should be owned by the client who created it. When the client deletes the **Arguments Object**, it should release any resources it is holding.

Known Uses

The X Window System's drawing operations use GraphicsContexts to package up a large number of arguments such as the font, colour, line width, and clip region (Scheifler & Gettys 1986). Smalltalk's Point and Rectangle objects can be seen as **Arguments Objects** which package up two or four integer arguments to describe points or rectangles (Beck 1995). Smalltalk also uses Message objects which record the arguments and name of a message which has caused an error (Goldberg & Robson 1983).

Common Variant: Event Object

How can you pass many arguments in response to user events?

Interaction objects (known as widgets or controls) in window systems and GUI frameworks need to respond to user events. Most events, such as mouse drags or key clicks, have a large number of associated parameters to describe details of interactions, such as the mouse position, mouse button state, and modifier key settings. If these events are delivered simply by messages, with each event parameter passed as an argument, each widget method would require a very large number of arguments.

Therefore: *Make an **Event Object** to represent the event, and to contain all the arguments which describe it.*

Methods which handle user actions can take only a single **Event Object** argument, rather than a large number of parameters to accept each argument individually. The forces and consequences of **Event Object** are basically the same as those for **Arguments Object**; however event-handling is sufficiently common that this variant is worth recognising in its own right. **Event Object**s are also good generic examples of **Argument Objects** because they model an important concept from the domain of user interaction frameworks.

MacApp (Schmucker 1986) and the X (Scheifler & Gettys 1986) Window System all use **Event Objects** to package arguments describing user events.

Related Patterns

The following two patterns describe how **Arguments Object** can be applied in particular situations. The number of messages in a protocol can be reduced by using a **Selector Object (2)**. Constant or slowly-varying arguments can be factored out by a **Curried Object (3)**, which introduces an intermediary object to elaborate a protocol.

2 Selector Object

How can you simplify a protocol where several messages differ mainly in their names?

Some protocols include several messages which perform the same underlying function. For example, a graphical View object provides many messages which draw graphical objects. These messages take substantially the same arguments and differ in the fine details of the precise function they perform — in the case of the View, whether to draw a rectangle, a filled rectangle, or an oval.

Protocols where many messages perform similar functions are often difficult to learn and to use, especially as the similarity is often not obvious from the protocol's documentation. Because the messages are conceptually closely related, they will often need to be maintained as a group, which will require changing a number of method implementations in servers and many different message sends in clients.

Therefore: *Make a single message taking an object representing the message selector as an extra argument.*

Remove the similar messages from the protocol, and replace them with a single message which takes a **Selector Object** as an additional argument. This message should perform the essential function performed by the messages from the original protocol, and use the **Selector Object** argument to discriminate between the functions in detail (typically using double dispatching (Gamma et al. 1994)). Change the protocol's clients and servers to use the new protocol.

In some cases, the **Selector Object** can be a very lightweight object, such as a symbol or enumeration, which is used to determine the fine details of the function to be performed. In other cases, the **Selector Object** can have substantial state and behaviour of its own. If you already have an **Arguments Object**, the **Selector Object** can be often be incorporated into it as an extra attribute. Alternatively, you can build a hierarchy of **Arguments Object** classes, with one class for each **Selector Object** which can be passed in the protocol.

Example

Consider again the original protocol provided by a graphical View object:

```
drawRectangleFrom: topLeft to: bottomRight colour: colour.
fillRectangleFrom: topLeft to: bottomRight colour: colour.
drawOvalFrom: topLeft to: bottomRight colour: colour.
```

Introducing an **Arguments Object** (a Graphic) can simplify the arguments passed in the protocol, as descibed above:

```
drawRectangle: aGraphic.
fillRectangle: aGraphic.
drawOval: aGraphic.
```

Each of these three messages performs essentially the same function — drawing on a View. The details of the function (whether to draw a rectangle outline, a filled rectangle, or an oval) are encoded in the message selector. We have an **Arguments Object** (the Graphic), so we can apply the **Selector Object** pattern to incorporate the message selector into the **Arguments Object**. Graphic can be extended to record a type, which selects either outlined rectangles, filled rectangles, or ovals to be drawn. View's protocol now contains only a single message with a single argument, draw: graphic, modelling the one essential function of the whole protocol.

```
"Create a Graphic"
   rectangle := Graphic type: #rectangle  from: 10@10 to: 20@20 colour: #red.
"Draw it"
   view draw: rectangle.
"And so on..."
   filledRectangle := Graphic type: #filledRectangle  from: 30@10 to: 40@20 colour: #blue.
   view draw: filledRectangle.
   oval := Graphic type: #oval  from: 50@10 to: 60@20 colour: #green.
   view draw: oval.
```

As an alternative, a family of Graphic subclass can be used, rather than an single class with a type attribute, as shown in Figure 5.

```
"Create a Graphic"
   rectangle := RectangleGraphic from: 10@10 to: 20@20 colour: #red.
"Draw it"
   view draw: rectangle.
"And so on..."
   filledRectangle := FilledRectangleGraphic from: 30@10 to: 40@20 colour: #blue.
   view draw: filledRectangle.
   oval := OvalGraphic from: 50@10 to: 60@20 colour: #green.
   view draw: oval.
```

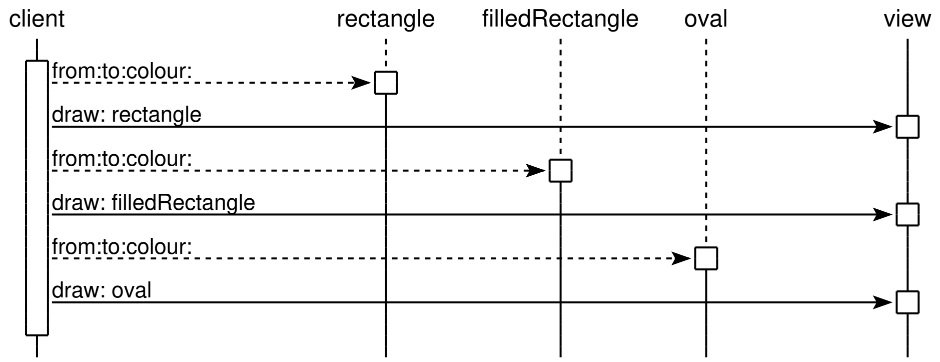

Figure 5: Selector Object

Consequences

The **Selector Object** pattern is a refinement of **Arguments Object (1)**, and its benefits and liabilities are similar. The tradeoff is slightly different — **Selector Object** trades off N similar messages at the server for 1 message with an extra argument plus the **Selector Object** and its creation message. When this factorisation results in **Selector Objects** which have meaning in the domain, the protocol will be smaller and easier to modify, and also easier to learn and to use. **But:** clients need to create the **Selector Object**, and servers need some mechanism to select the actual function the message will perform.

Known Uses

Selector Objects are often used to build object oriented interfaces to existing file or graphics systems. For example, VisualWorks uses symbols representing file access modes as arguments to messages to manage files (Par 1994). Many OO graphics systems, again including VisualWorks and Interviews, provide Geometric or Glyph objects which combine the **Selector Object** and **Arguments Object (1)** patterns.

Related Patterns

Lightweight **Selector Objects** can often be **Flyweights** (Gamma et al. 1994).

3 Curried Object

How can you simplify an extremely complicated protocol?

Over the course of a program, objects exchange messages, and the objects passed as arguments to these messages are usually different every time. Sometimes, an object will receive a series of messages where one or more arguments are constant. For example, a text editor will often draw a number of different strings in exactly the same font size, typeface, and colour. Arguments may also be sent in sequence, so that the argument to one message can be predicted from the corresponding argument of a previous message — the text editor will draw the strings with the same left margin, but each offset one line lower on the screen.

These kinds of arguments increase the complexity of a protocol. The protocol will be difficult to learn, as programmers must work out which arguments must be changed, and which must remain constant. This information is not explicitly represented in the protocol, and often not provided by standard documentation. The protocol will be difficult to use, as clients must cache constant arguments between sends and compute the values of slowly-varying arguments.

Therefore: *Send simpler messages to an intermediary (a **Curried Object**) which elaborates them within its context.*

A **Curried Object** stores the constant or slowly varying arguments from the original protocol, and provides a simpler protocol with these arguments eliminated. A **Curried Object** stores the original server object and forwards messages to it, passing along the stored arguments and updating the slowly varying arguments.

To use a **Curried Object**, change clients so that rather than sending messages to the server, they create a **Curried Object**, initialise it as required, and send messages to it. The server's protocol can be extended to create and initialise a **Curried Object** which refers to that server. The original protocol can either remain publicly available in the server, or be restricted to the **Curried Object**.

Example

Consider drawing lines of text on a graphical view, as shown in Figure 6.

```
"Set up arguments"
   font := Font named: 'Times'.
   origin := (20 @ 30).
   offset := (0 @ (font textHeight)).
"Draw text"
   view drawString: 'This is an example' at: origin font: font.
   view drawString: 'to illustrate' at: (origin + offset) font: font.
   view drawString: 'the problem' at: (origin + (offset * 2)) font: font.
```

The drawString:At:Font message takes three arguments. The first argument, the string to draw, is different for each message sent; the second argument, the point at which to draw the string, varies according to an arithmetic progression; and the third argument, the font to use, is constant.

This protocol can be simplified by introducing a **Curried Object**. The **Curried Object** (which we will call a Pen) requires three variables to hold the View server object, and the origin and font arguments of the drawString:At:Font message. Pen's protocol will include messages to read and write these variables, and a single argument message drawString to draw a string and advance to the next line (updating the origin variable). The View object needs a single argument message pen which returns a new Pen object associated with the View. As illustrated in Figure 7, using a Pen, the example above becomes:

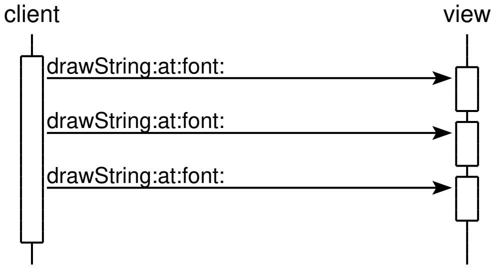

Figure 6: A Complicated Protocol

"Create a Pen"
pen := view pen at: (20@30) offset: (0 @ (font textHeight)) font: (Font named: `'Times'`).
"Use the Pen to draw strings"
pen drawString: `'This is an example'`.
pen drawString: `'to illustrate'`.
pen drawString: `'curried objects'`.

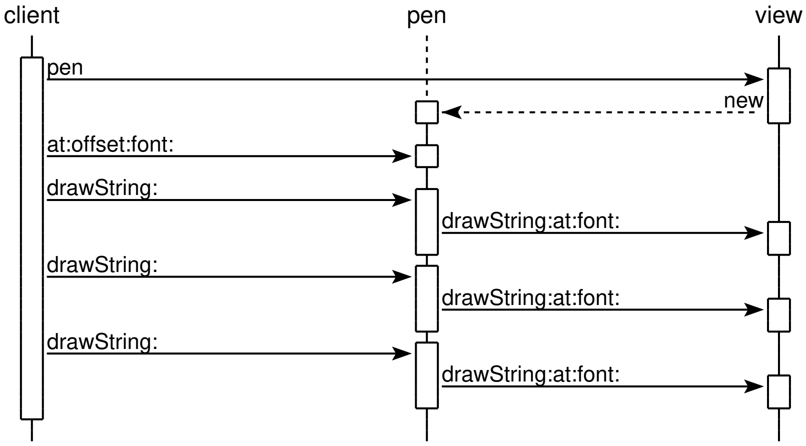

Figure 7: Curried Object

Consequences

Curried Object is similar to **Arguments Object (1)**, and shares most of the benefits and liabilities of that pattern; however, **Curried Object** requires fewer changes to existing servers. Like **Arguments Object (1)**, **Curried Object** is most effective when it reifies objects from the program's domain. **But: Curried Object** displaces the receiver of the protocol, while **Arguments Object (1)** does not. **Curried Object** thus introduces *action at a distance*, because messages sent to one object (the **Curried Object**) are acted upon by another object (the original server). A **Curried Object** is at the same level of abstraction as its server, and acts as an alias for the server. Programmers need to know about both the original server and the **Curried Object**, and understand the distinctions between them, in particular, which messages to send to which object.

This pattern is called *Curried Object* because the underlying mechanism is partial function application†, colloquially known as *currying* after the mathematician Haskell B. Curry (Bird & Wadler 1988). Currying is supported directly in several functional programming languages, including the Haskell language that is also named after

† Partial function application means that you can provide only some of the arguments to a function call, and the result is a new function which can be called later with the rest of the arguments. For example, given a two argument function "plus a b" that sums its arguments, we can define a one argument function that adds one to its argument by calling plus with just its first argument "fun 'addone = plus 1".

Curry (Hudak, Jones & Wadler 1992). The name *Curried Object* also suggests that this pattern is a little spicy and exotic, and probably not for the beginner.

Known Uses

Iterators (Gamma et al. 1994) are the most common kind of **Curried Object**, as they are used in many languages and class libraries (Goldberg & Robson 1983, Stroustrup 1997, Booch 1994). An iterator's server is a collection object, and the iterator maintains a slowly varying index into the collection. Many graphics systems use **Curried Objects** whose servers are views, as in the example above. For example, VisualWorks uses GraphicsContext objects (Par 1994) and Smalltalk/V uses Pen objects (Lalonde 1994). VisualWorks also includes MessageSend objects, a curried version of the Message **Arguments Object (1)**. MessageSend inherits from Message, and adds an extra variable to store the message's receiver, allowing a message to be sent without an explicit reference to the ultimate receiver object.

Related Patterns

Arguments Object (1) can provide a less radical alternative to **Curried Object**. The original server can act as an **Abstract Factory** (Gamma et al. 1994) to create the **Curried Object**. A **Curried Object** can be similar to an object-level **Adaptor** (Gamma et al. 1994), but where an adaptor allows an object to conform to an existing protocol, a **Curried Object** introduces a new, simpler protocol. The **Command** (Gamma et al. 1994) pattern can also be seen as refining **Curried Object**. A **Command** object stores a message selector, message arguments, and an object to receive the message, but often supports other behaviour, such as undo. The **Accumulator** (Yelland 1996) pattern is a variant of **Curried Object** which simplifies the protocol used to create objects. The **Type-Safe Session** (Pryce 1999) pattern is a **Curried Object** which emphasises type safety.

Patterns about Results

Many messages ask questions of the objects to which they are sent, and the results of these messages are the answers to these questions. The following three patterns describe how objects can simplify the asking and answering of questions.

4 Result Object

How can you return complex answer to a question?

You have a long or important computation performed by a server object, and you wish to retain the results of the computation. Perhaps the computation returns more than one object, or the result is needed at several times or places throughout the program, or you need to keep information about how the result was obtained. For example, consider a MetricCalculator for a programming environment that calculates software metrics for a system under development. Calculating the metrics is a long computation and needs to return values for a number of different metrics. The programming environment needs to keep the values for the different metrics together, to record the date, time, and configuration for which the metrics were generated, and to store the values to track the evolution of the system over time.

The client object could cache the results itself, but this increases the client's complexity, as the resulting caching code will obscure the main application logic, making it difficult to read and modify. Alternatively, the server object could cache and store the results, but this has similar problems, in that the caching code pollutes the implementation of the domain computation. This also complicates the server's protocol, since it must be able to return both previous and current results.

Therefore: *Return a **Result Object** that is the whole answer to the question.*

Make one variable in the **Result Object** for each result to be returned, and provide the usual accessor and initialisation messages. If additional information about the result is required, store this in the **Result Object** also, again with appropriate accessors. Modify the server to create and return a **Result Object**, and the server's clients to retrieve the results from the **Result Object**.

Example

Consider a MetricCalculator object for calculating software metrics, illustrated in Figure 8.

```
"Make a MetricCalculator"
  m := MetricCalculator for: anObject.
"Use it to compute some metrics"
  m computeSizeOfInterface.
  m computeNumberOfInheritedMethods.
  m computeNumberOfOverriddenMethods.
```

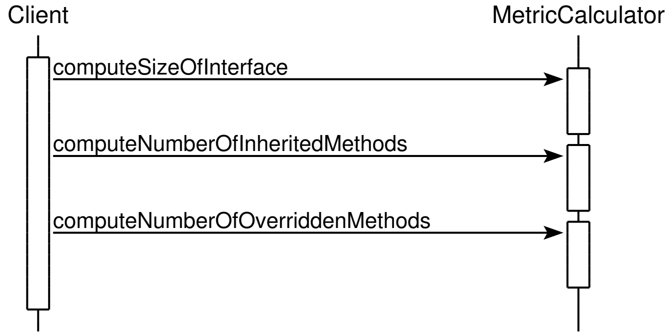

Figure 8: Metric Calculator

The various compute messages traverse the target object's inheritance hierarchy and compute the metrics. The hierarchy is traversed whenever an individual metric is required, and the traversal is repeated if the metric is needed again. Since each metric requires exactly the same traversal of the inheritance hierarchy, this is really a single computation returning multiple results — the various different metrics.

This protocol can be improved by introducing a **Result Object**. All the metrics can be calculated in one traversal, and a **Result Object** (called a MetricReport) returned. Individual metric values can be retrieved from the MetricReport object. The MetricReport object can also store ancillary information about the metric calculation, such as the configuration and date for which the metrics were calculated. Figure 9 illustrates how the above example can be redesigned to use a MetricReport:

```
"Make a MetricCalculator"
  m := MetricCalculator for: anObject.
"Compute the metrics, producing a MetricReport"
  metricReport := m computeMetrics.
"Access metrics from the MetricReport"
  metricReport sizeOfInterface.
  metricReport numberOfInheritedMethods.
  metricReport numberOfOverriddenMethods.
```

Consequences

A **Result Object** is quite like a **Curried Object (3)** and shares many of the benefits and liabilities of that pattern, except that a **Result Object** substitutes for a message's result, while a **Curried Object (3)** substitutes for a message's receiver. A **Result Object** trades the server's protocol size for an extra object — M messages to the server are replaced by 1 message to return a **Result Object** plus M messages to retrieve the result values. The resulting protocols are usually easier to learn and to read, and with reduced coupling, are easier to change. **Result Object** is particularly powerful when the new **Result Object** corresponds to a concept from the domain. **But:** for the client object programmer, **Result Objects** are more difficult to use than server-side caching, because the client needs to extract the actual results from the **Result Object**. Similarly, for the server object programmer, **Result Objects** require more work than client-side caching.

As with **Arguments Object**, **Result Object** introduces a new object into the program, and in languages without garbage collection (such as C++) the **Result Object's** lifetime must be managed explicitly. A **Result Object**

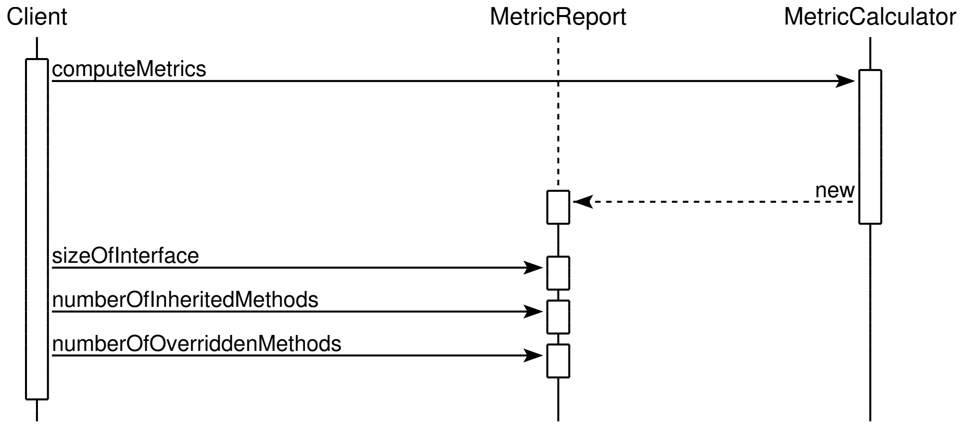

Figure 9: Result Object

should be owned by the client who received it as a result, rather than the server which created it. When the client deletes the **Result Object**, it should release any resources it is holding.

Known Uses

The VisualWorks Date class returns a **Result Object** (called TimeStamp) to package together the current date and time. As well as simplifying client code, this also avoids the problems which would occur if the time was returned at one second before midnight, and the date one second afterwards. Martin Fowler discusses similar **Result Objects** called TimePoints (Fowler 1997). An expert system used in Telecoms capacity planning used a **Result Object** to package the decisions it returned with the logic supporting the decisions. The **Result Object** allowed the system's users to check the supporting logic to verify that the decisions were being made appropriately. The *Self Programmer's Reference Manual* describes how **Result Objects** can be used in Self to return multiple values from messages (Agesen, Bak, Chambers, Chang, Hölzle, Maloney, Smith, Ungar & Wolczko 1995).

Common Variant: Error Object

How can you return information about an error in a computation?

Sometimes computations produce errors, rather than return results. When an error occurs, programmers often need to store information about the error. Later, when handling the error, programmers need access to this information to log the error, take corrective action, or present an error message to the user. Information about errors can be stored in the server object, and retrieved explicitly by the client. This increases the server's complexity, reduces its cohesion, increases coupling between client and server, and increases the complexity of the server's protocol.

Therefore: *Return an **Error Object** that contains all the arguments which describe the error.*

An **Error Object** is a **Result Object** used when a computation finishes with an error rather than a result. **Error Object** is a refinement of **Result Object**, so the forces and consequences of **Error Object** are very similar to those of **Result Object**. **Error Object**s are sufficiently common that this variant is worth describing in its own right.

VisualWorks includes SystemError objects, **Result Objects** that package together return codes and identifying arguments from errors occurring outside the system (Par 1994). Some programming languages, including Java, C++, and Common Lisp, provide special support for **Error Objects** called Exceptions. These **Error Objects** allow programmers to package up all the information related to a particular error when it occurs, and then retrieve that information when handling the error.

Related Patterns

If the question could be answered by a concurrent computation, try **Future Object (5)**. If the question could be answered easily now, but the answer may never be needed, try **Lazy Object (6)**. A **Memento** (Gamma et al. 1994) is a specialised **Result Object** which opaquely checkpoints an object's entire state so that the object can be completely restored later.

5 Future Object

How can you answer a question while you think about something else?

Sometimes you need to ask a question, and then do something else while waiting for the answer to arrive. For example, a programming environment may need to respond to its user interface while a MetricCalculator computes the metrics for the latest release. In these cases, the computation's result must be returned to the program eventually, but it is not needed immediately. If the computation can be performed independently from the rest of the program — that is, if the computation does not modify objects which the rest of the program depends upon and vice versa — it should be possible to compute the result in a concurrent thread.

Unfortunately, managing concurrent threads is quite difficult in practice. Because the computation's result is eventually required by the program, a new thread cannot be started and left to its own devices. Rather, the result somehow must be returned to the program when the computation finishes. This will increase the complexity of the object chosen to control the thread, and reduces its cohesion by contaminating it with process management issues. In practice, programmers often ignore the issue and compute the result immediately, delaying the program's execution until the computation is complete, and consequently reducing the program's responsiveness and performance.

Therefore: *Return a **Future Object** that represents the eventual result of a concurrent computation.*

A **Future Object** is a **Result Object (4)** which computes an answer in a concurrent thread, based on initial information supplied by the original server (Halstead, Jr. 1985). The **Future Object** should receive the computation's arguments from the server, and then cache any additional information that it needs but which may change once the main computation resumes. The **Future Object** should handle the thread management — creating a new thread to perform the computation, and storing the results when the thread completes. The **Future Object** should also provide synchronisation — clients which access the **Future Object** while its associated thread is running should be blocked until the thread completes.

The **Future Object** can also be used to control the thread performing the computation. This control can be direct — the **Future Object** can provide an interface to control the thread, or indirect — if the **Future Object** is deleted before its computation completes, the unneeded computation can also be deleted.

Example

Consider the metricReport example from the **Result Object (4)** pattern. A **Future Object** could implement a metricReport object with exactly the same interface but with a different performance profile, as shown in Figure 10. Using a **Result Object**, the program will wait while the computation is performed. Using a **Future Object**, the computation will be performed in a concurrent thread (started by the computeMetrics message)). The program will wait if the computation has not finished when the result is required, that is, when the program next accesses the metricReport **Future Object** — in Figure 10, when the sizeOfInterface message is sent. If the creation of the **Future Object** and its first access are sufficiently separate, the metrics computation will not delay the main thread.

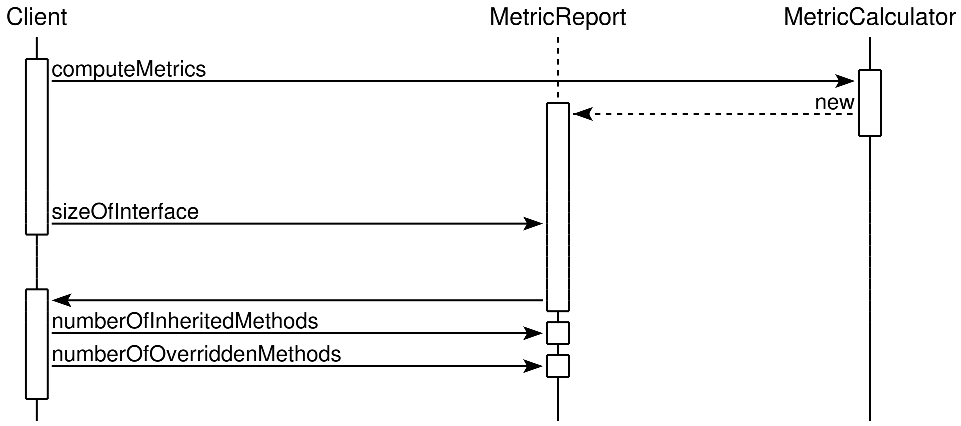

Figure 10: Future Object

Consequences

A **Future Object** has exactly the same protocol as a **Result Object (4)**, and so has similar benefits of increased readability and decoupling, and decreased complexity. In addition, a **Future Object** separates concurrency issues from clients' and servers' domain code, so the implementation of concurrency is easier to change. **But:** because a **Future Object** must manage the concurrency, it will have a larger overhead than a standard **Result Object**. **Future Object** obviously requires that the language or runtime system supports concurrency. A **Future Object** will also be more difficult to write than a **Result Object**, although some languages are sufficiently flexible that a single generic **Future Object** can be written once and reused as necessary (Foote & Johnson 1989). A **Future Object** cannot avoid the intrinsic problems imposed by concurrency, such as deadlock, exclusion, race conditions, and so on, so the program will become harder to debug, and its performance will become less predictable (Lea 1997). Also, because of the concurrency, this pattern should be applied only when the concurrent computation is completely independent of the rest of the program.

Known Uses

Future Objects are quite common in concurrent, parallel, and distributed computing (Yonezawa & Tokoro 1987). They were introduced in Multilisp (Halstead, Jr. 1985) and have been used in Actor languages (Lieberman 1987), Smalltalk (Foote & Johnson 1989, Yokote & Tokoro 1987) and MUSHROOM (Hopkins & Wolczko 1989) amongst many other systems (Lea 1997).

Related Patterns

A **Future Object** can be seen as a **Proxy** (Gamma et al. 1994) for an object which is being computed in a concurrent thread. **Future Object** has been described briefly as part of the **Active Object** pattern (Schmidt & Cranor 1997).

6 Lazy Object

How can you answer a question that is easy to answer now, but that may never be asked?

Some computations can best be performed immediately but the computation's result may never be needed. For example, the data needed by the MetricCalculator may only be accessible for the current version of the software being developed, but it is unlikely that the user would want metrics calculated for every intermediate version. Often this kind of computation depends on information which is available now, but may not be available in the future.

The simplest solution to this problem is for the client to ask the question every time, and then store the results until they are needed. If most of the results are not needed, this will cause a large amount of unneeded computation, and also complicate the client's code, making it harder to write. Alternatively, the server object answering the question could compute and cache all the results, with similar problems of efficiency and increased server complexity. The client could postpone asking the question until the answer is required, but, when the question is asked, the information the answer depends upon may be lost.

Therefore: *Return a **Lazy Object** that can answer the question later, if necessary.*

A **Lazy Object** is a **Result Object (4)** which does not start its computation until the answer is requested. The laziness is managed within the **Lazy Object**, rather than by the clients or server. A server object can initialise and return a **Lazy Object** in exactly the same way it would return a **Result Object**. The server passes the computation's arguments to the **Lazy Object**, which stores this information. The **Lazy Object** also needs to cache any other information that the computation needs, and which may change between the time it is created and the time it is used. When it is accessed, the **Lazy Object** cannot return the result of the computation, because the computation has not yet been carried out. Rather, the **Lazy Object** should perform the computation and cache the results the first time it is used. Future requests can be serviced from the cache.

Example

Consider again the metrics calculator example from **Result Object (4)**. A **Lazy Object** could be used as the metricReport, changing the performance characteristics but not the interface, as shown in Figure 11. Using a **Lazy Object**, the metrics computation will be performed when the first metric is requested — in the example, when the sizeOfInterface message is sent. If the result is never needed, the **Lazy Object** can be deleted or garbage collected, and the metrics will never be calculated.

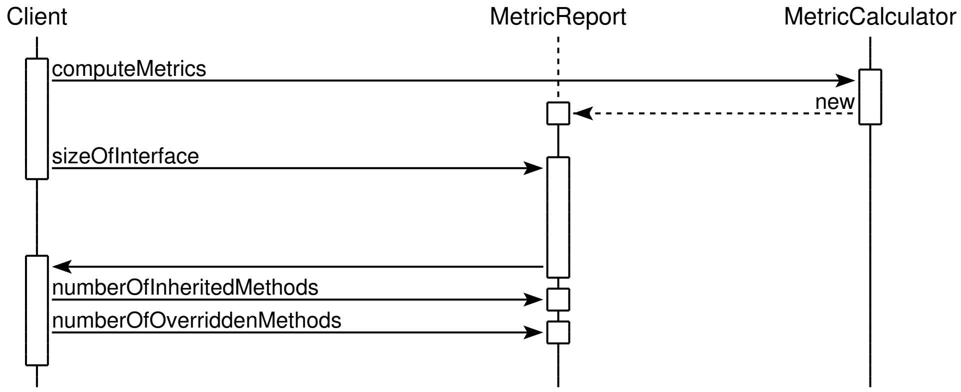

Figure 11: Lazy Object

Consequences

A **Lazy Object** is a refinement of a **Result Object**, and shares the benefits and liabilities of that pattern. In addition, a **Lazy Object** avoids the overhead of unnecessary computations by calculating only results that are actually required. Since a **Lazy Object**'s own protocol is exactly the same as that of a corresponding **Result Object**, the choice to use (or not use) lazy evaluation can be completely hidden from any client objects. Similarly, because the **Lazy Object** manages the evaluation itself, the server is mostly insulated from the details of the laziness. **But:** these gains in efficiency must be traded off against the intrinsic problems of lazy computation — the need to identify and cache information which the computation depends upon and which might be changed by the rest of the program, and the need to ensure that the computation (if eventually performed) does not itself have side effects on the rest of the program (Bird & Wadler 1988, Abelson & Sussman 1985). A **Lazy Object** may make program performance difficult to predict, and the program difficult to debug, because it is not easy to determine when (or if) the calculation is actually carried out. Most real world domains are *not* lazy, so introducing laziness can reduce the fidelity of the domain model embodied by the program.

Known Uses

The MUSHROOM system implements a generic **Lazy Object** (Hopkins & Wolczko 1989). The LOOM virtual memory system for Smalltalk used **Lazy Objects** called *leaves* to represent objects which were swapped out into secondary storage (Kaehler & Krasner 1983). *Design Patterns* describes how ET++ uses **Lazy Objects** called *virtual proxies* to represent large images which may not need to be displayed (Gamma et al. 1994).

Related Patterns

A **Lazy Object** can be seen as a **Proxy** (Gamma et al. 1994) for an object which hasn't been computed yet. Ken Auer and Kent Beck have described many similar patterns for optimising Smalltalk programs (Auer & Beck 1996).

The **Result Object (4)**, **Future Object (5)**, and **Lazy Object (6)** patterns are distinguished by then times when the computation is started, and when the result is returned to the main thread. A **Result Object** starts the computation and returns the result as soon as it is created, a **Future Object** starts a computation as soon as it is created, but returns the result when it is first accessed, and a **Lazy Object** both starts the computation and returns the result when it is first accessed.

References

Abelson, H. & Sussman, G. J. (1985): *Structure and Interpretation of Computer Programs*, MIT Press and McGraw-Hill.

Agesen, O., Bak, L., Chambers, C., Chang, B.-W., Hölzle, U., Maloney, J., Smith, R. B., Ungar, D. & Wolczko, M. (1995): *The Self Programmer's Reference Manual*, 4.0 edn, Sun Microsystems and Stanford University.

Auer, K. & Beck, K. (1996): Lazy optimization: Patterns for efficient smalltalk programming, in *Pattern Languages of Program Design*, Vol. 2, Addison-Wesley.

Beck, K. (1995): Parameters object. Email message sent to the patterns-digest list.

Beck, K. (1997): *Smalltalk Best Practice Patterns*, Prentice-Hall.

Bird, R. & Wadler, P. (1988): *Introduction to Functional Programming*, Prentice-Hall.

Booch, G. (1994): *Object Oriented Analysis and Design with Applications*, second edn, Benjamin Cummings.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. (1996): *Pattern-Oriented Software Architecture*, John Wiley & Sons.

Cunningham, W. (1995): About the portland form. `http://c2.com/ppr/about/portland.html`.

Foote, B. & Johnson, R. E. (1989): Reflective facilities in Smalltalk-80, in *OOPSLA Proceedings*.

Fowler, M. (1997): *Analysis Patterns*, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. E. & Vlissides, J. (1994): *Design Patterns*, Addison-Wesley.

Goldberg, A. & Robson, D. (1983): *Smalltalk-80: The Language and its Implementation*, Addison-Wesley.

Halstead, Jr., R. H. (1985): Multilisp: A language for concurrent symbolic computation, *ACM Transactions on Programming Languages and Systems* **7**(4), 501–538.

Hopkins, T. P. & Wolczko, M. (1989): Writing concurrent object-oriented programs using Smalltalk-80, *The Computer Journal* **32**(4).

Hudak, P., Jones, S. P. & Wadler, P. (1992): Report on the programming language haskell, *SIGPLAN Notices* **27**(5). Revised version at `http://www.haskell.org`.

Kaehler, T. & Krasner, G. (1983): LOOM–large object-oriented memory for Smalltalk-80 systems, in G. Krasner, ed., *Smalltalk-80: Bits of History, Words of Advice*, aw, chapter 14.

Lalonde, W. (1994): *Discovering Smalltalk*, Benjamin/Cummings.

Lea, D. (1997): *Concurrent Programming in Java*, Addison-Wesley.

Lieberman, H. (1987): Concurrent object-oriented programming in Act 1, in *Object-Oriented Concurrent Programming* (Yonezawa & Tokoro 1987).

Noble, J. (1998): Classifying relationships between object-oriented design patterns, in *Australian Software Engineering Conference (ASWEC)*, pp. 98–107.

Par (1994): *VisualWorks Smalltalk User's Guide*, 2.0 edn.

Parnas, D. L., Clements, P. C. & Weiss, D. M. (1985): The modular structure of complex systems, *IEEE Transactions on Software Engineering* **11**(3).

Perlis, A. (1982): Epigrams on programming, *ACM SIGPLAN Notices* **17**(9).

Pryce, N. (1999): Abstract session — an object structural pattern, in *Pattern Languages of Program Design*, Vol. 4.

Scheifler, R. W. & Gettys, J. (1986): The X Window System, *ACM Transactions on Graphics* **5**(2).

Schmidt, D. C. & Cranor, C. D. (1997): Active object: An object behavioral pattern for concurrent programming, in *Pattern Languages of Program Design*, Vol. 2, Addison-Wesley.

Schmucker, K. (1986): MacApp: an application framework, *Byte* **11**(8).

Stroustrup, B. (1997): *The C++ Programming Language*, 3rd edn, Addison-Wesley.

Yelland, P. M. (1996): Creating host compliance in a portable framework: A study in the use of existing design patterns, in *OOPSLA Proceedings*.

Yokote, Y. & Tokoro, M. (1987): Concurent programming in ConcurrentSmalltalk, in *Object-Oriented Concurrent Programming* (Yonezawa & Tokoro 1987).

Yonezawa, A. & Tokoro, M. (1987): *Object-Oriented Concurrent Programming*, M.I.T. Press.

Zimmer, W. (1994): Relationships between design patterns, in *Pattern Languages of Program Design*, Addison-Wesley.