Arguments and Results

James Noble
MRI, School of MPCE,
Macquarie University, Sydney.
kjx@mri.mq.edu.au

October 20, 1997

Abstract

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 using these patterns, programs and designs can be made more simple, more general, and more easy 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. Using an object's protocol, other objects in the program can use the object as a *server*, thus accessing the behaviour the object provides. Similarly, an object can act as a *client* to other objects, in turn using their protocols to access their services.

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 can receive, and the results objects return in response to the messages. These patterns do not attempt to describe novel techniques, rather, they present well-established solutions for object oriented design. These patterns should be useful for introducing the techniques to novice programmers, and for more experienced programmers, should illustrate when particular techniques are applicable and their relative strengths and weaknesses.

These patterns are interrelated, but they do not form a whole language. Rather, the patterns are two fragments (**Patterns about Arguments** and **Patterns about Results**) which may one day form part of a larger pattern language. Each of the fragments has the same structure (see Figure 2) with one general pattern, and a couple of more specific patterns which refine the general pattern to handle specialised contexts. Figure 2 also illustrates two relationships between the patterns. One pattern can *refine* 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 [26].

Forces

Each of these patterns resolves a number of different forces, and some conflicting patterns resolve similar problems in different ways. 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 complexity to be handled once in *server* objects, rather than many times in every *client* object which uses the servers.

Pattern	Problem	Solution
Arguments Object	How can you simplify a complex protocol that has a regular argument structure?	Make an Arguments Object to capture the common parts of the protocol.
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 manage a difficult answer to a difficult question?	Make a Result Object the whole answer to the question.
Future Object	How can you answer a question while you think about something else?	Make a Future Object which computers the answer in parallel.
Lazy Object	How can you answer a question that is easy to answer now, but that may never be asked?	Make a Lazy Object which can answer the question later, if necessary.

Figure 1: Summary of the Patterns

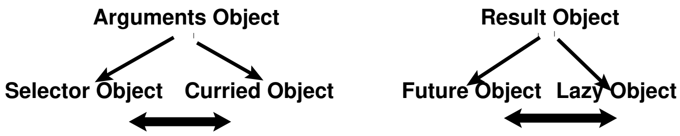


Figure 2: The Structure of the Language

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[1], 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. To quote Alan Perlis: *In the long run every program becomes rococo — then rubble.* [20]. If these patterns are applied injudiciously they will accelerate this process.

[1]This is not limited solely to these patterns. Many other patterns have this effect, including those in *Design Patterns* [9] and *Smalltalk Best Practice Patterns* [5], as do Parnas's criteria for program decomposition [19].

Form

The patterns are written in electric modified Portland form. 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 the a description of the solution in detail. Then follows an example of using the pattern, the patterns consequences (benefits first, a boldface **But:**, then the liabilities) and finally some known uses and related patterns.

Patterns about Protocols

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 protocols.

1 Arguments Object

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

Some objects have large and complex protocols with a regular internal structure. For example, a graphical View object will provide 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* [20].

Therefore: *Make an* **Arguments Object** *to capture the common parts of the protocol.*

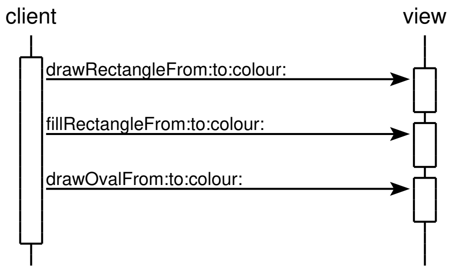
In its simplest form, an **Arguments Object** should have one variable for each argument to be eliminated from the protocol, and the usual messages to access and update its variables. Change the protocol and its implementations to accept a single **Arguments Object** in place of the eliminated arguments,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

Example

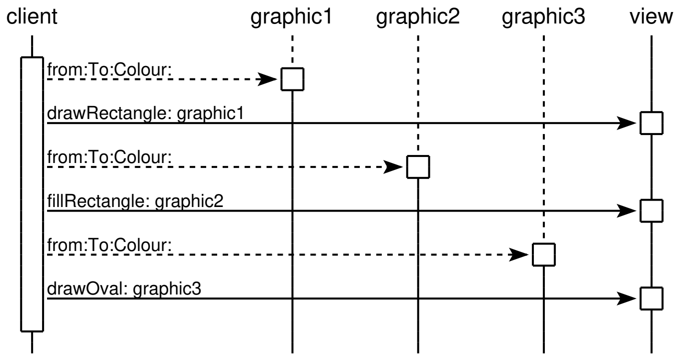
Consider the protocol provided by a graphical View object:

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

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.



"Create a Graphic"
   g := Graphic from: topLeft To: bottomRight Colour: colour.
"Draw it"
   view drawRectangle: g.



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 fewer arguments plus a new **Argument Object** which requires N arguments to create it. 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 those objects which fundamentally rely on the changed protocol. Ideally, an **Arguments Object** will explicitly reveal an object from the application domain. **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. As with all these patterns, this pattern introduces an additional object into the design, requiring modifications to the program and increasing runtime space and time costs.

Known Uses

MacApp uses Event objects to package the arguments sent to widgets in response to user actions [24]. 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 [22]. 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 [4]. Smalltalk also uses Message objects which record the arguments and name of a message which has caused an error [10].

Related Patterns

The following two patterns describe how **Arguments Object** can be applied in particular situations. The number of message names 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 then with a single message which takes the **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 multimethods or double dispatching [9]). 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 only 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 folded into it, resulting in a **Message Object**.

Example

Consider the protocol provided by a graphical View object, which uses a Graphic **Arguments Object** as described above.

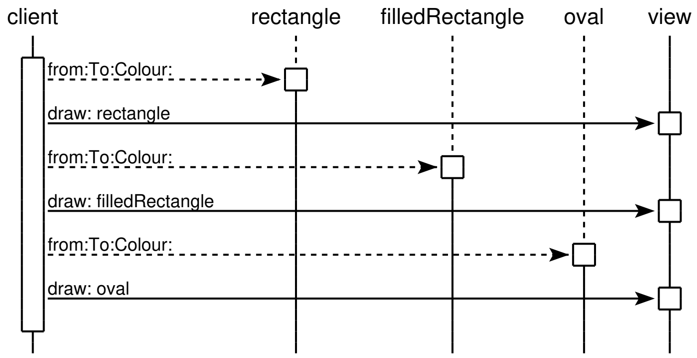
```
drawRectangle: aGraphic
fillRectangle: aGraphic
drawOval: aGraphic
```

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. In this example, 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, draw, modelling the one essential function of the whole protocol.

```
"Create a Graphic"
  g := Graphic
     type: #rectangle  from: topLeft
        to: bottomRight colour: colour.
"Draw it"
  view draw: g.
```

As an alternative, a family of Graphic subclass can be used, rather than the type argument.

```
graphic := RectangleGraphic
                from: 10@10 to: 20@20 colour: #red.
   view draw: graphic.
```



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 [18]. Many OO graphics systems, again including VisualWorks, provide Geometric or Graphic objects which combine the **Selector Object** and **Arguments Object (1)** patterns.

Related Patterns

Lightweight **Selector Objects** can often be **Flyweights** [9].

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 this object, passing along the stored arguments and updating the slowly varying arguments.

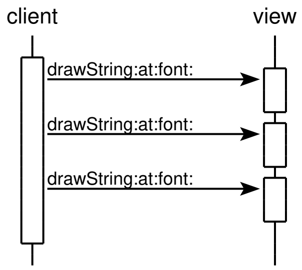
To use a **Curried Object**, change clients so that rather than sending messages to the server, they create a curried object, initialise it as required, and send messages to it. Protocol can be added to the server for creating and initialising a **Curried Object** which refers to that server. The original protocol can remain publicly available in the server, or it can be restricted to the **Curried Object**.

Example

Consider drawing lines of text on a graphical view:

```
font := Font named: 'Times'.
offset := (0 @ (font textHeight)).

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.
```

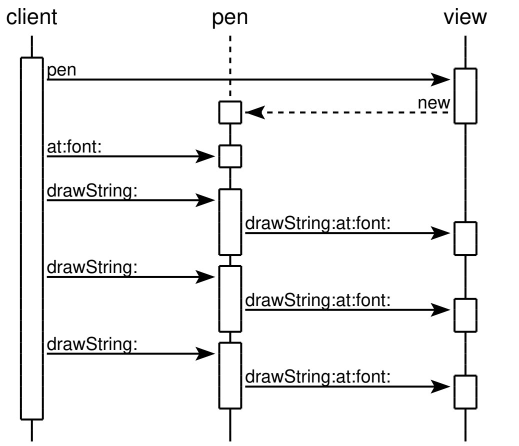


The drawString:At:Font message takes three arguments. The first argument, the string to draw, is different for each message sent; the second argument, the point at which to draw the string, varies according to an arithmetic progression; and the third argument, the font to use, is constant.

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. Using a Pen, the example above becomes:

```
pen := view pen at: origin font: (font named: 'Times').

pen drawString: 'This is an example'.
pen drawString: 'to illustrate'.
pen drawString: 'curried objects'.
```



Consequences

Curried Object is quite similar to **Arguments Object (1)**, and shares most of the benefits and liabilities of that pattern, however, it requires fewer changes to existing servers. **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 actually executed by another (the original server object). 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 [12]. The name *Curried Object* also suggests that this pattern is a little spicy and exotic, and probably not for the beginner.

Known Uses

Iterators are the most common kind of **Curried Object**, as they are used in many common languages and class libraries [9, 10, 6]. 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 [18] and Smalltalk/V uses Pen objects [15]. 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 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** [9] to create the **Curried Object**. A **Curried Object** can be similar to an object-level **Adaptor** [9], but where an adaptor allows an object to conform to an existing protocol, a **Curried Object** introduces a new, simpler protocol. The **Accumulator** [25] pattern is a variant of **Curried Object** which simplifies the protocol used to create objects. The **Type-Safe Session** pattern is a **Curried Object** which emphasises type safety [21].

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 store and manage a difficult answer to a difficult 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, 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 return both previous and current results.

Therefore: *Make a* **Result Object** *the whole answer to the question.*

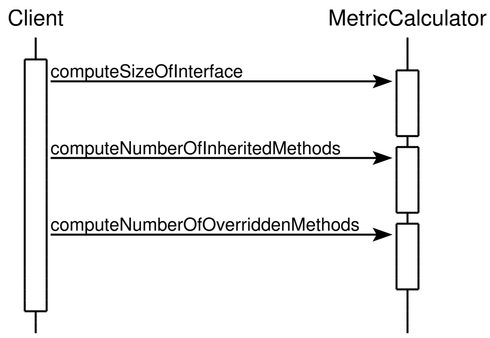
Make one variable in the **Result Object** for each value to be returned, and provide the usual accessor and initialisation messages — if additional information about the result is required, store this in the **Result Object** also. Provide the usual accessor messages so that this information can be retrieved from the **Result Object**. 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:

```
m := MetricCalculator for: anObject.

m computeSizeOfInterface.
m computeNumberOfInheritedMethods.
m computeNumberOfOverriddenMethods.
```

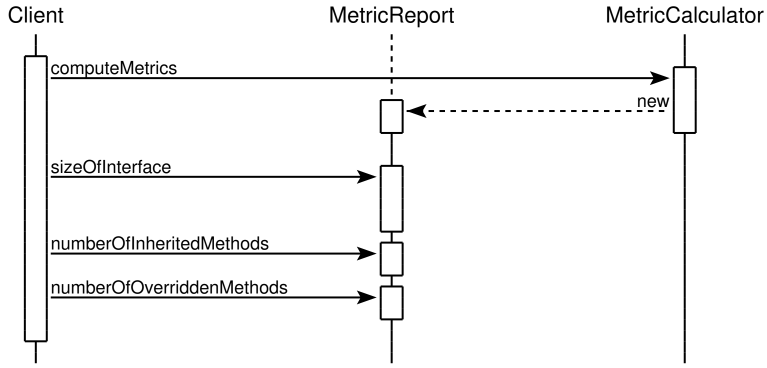


The various compute messages traverse the target object's inheritance hierarchy and compute metrics. The hierarchy is traversed whenever an individual metric is required, and repeated if the metric is needed again. Since each metric requires exactly the same traversal of the inheritance hierarchy, this 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. The MetricReport object can also store ancillary information about the metric calculation, such as the date the metrics were calculated. Using a MetricReport, the above example becomes:

```
metricReport := (MetricCalculator for: anObject) computeMetrics.

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 **Curried Object (3)** substitutes for a message's receiver, while a **Result Object** substitutes for a message's result. A **Result Object** trades the server's protocol size for extra objects — 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**. For the server object programmer, they similarly require more work than client-side caching.

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 [8]. 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 system's users could 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 [1].

Result Objects are often used to provide error handling (resulting in **Error Objects**). Representing errors with **Result Objects** allows the errors to be queued as they occur, and displayed later to the user. These **Result Objects** can also provide textual descriptions of the errors, and appropriate help information. For example, VisualWorks includes SystemError objects, **Result Objects** which package together return codes and identifying arguments from errors occurring outside the system [18].

Related Patterns

If the question can be answered in parallel, try **Future Object (5)**. If the question can be answered easily now, but the answer may never be needed, try **Lazy Object (6)**.

5 Future Object

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

Sometimes you need to ask a question, 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 parallel thread.

Unfortunately, managing parallel threads is quite difficult in practice. Because the computation's result is eventually required by the program, the thread cannot be started in parallel and left to its own devices. Rather, the result must be returned to the program when the computation finishes. If the server starts the parallel computation, then either the server or the computation's clients must extract the result from the thread. This will increase the complexity of the object chosen to have this responsibility, and reduces cohesion by contaminating it with process management issues. In practice, programmers often adopt the simplest solution, ignoring the issue and computing the result immediately, delaying the program's execution until the computation is complete, consequently reducing the program's responsiveness and performance.

Therefore: *Make a* **Future Object** *which computes the answer in parallel.*

A **Future Object** is a **Result Object (4)** which computes an answer in a parallel thread, based on initial information supplied by the original server [11]. The **Future Object** should receive the computation's arguments from the server, and then cache any information which may change after the main computation resumes. The **Future Object** should handle the thread management — creating a new thread to perform the computation, and extracting 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.

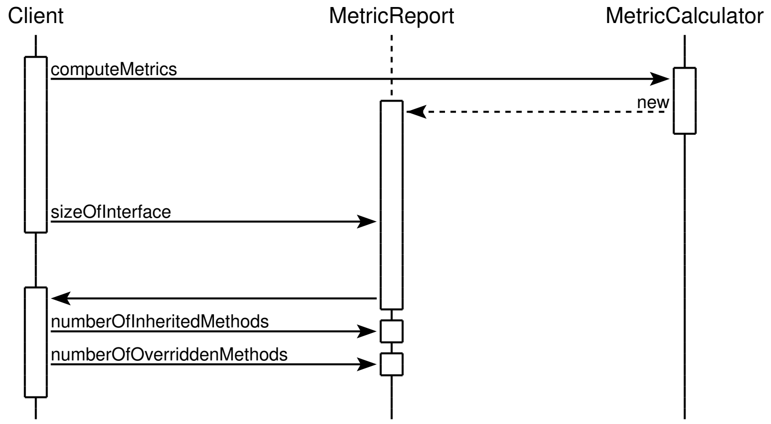
The **Future Object** can also be used to control the thread performing the computation. This control can be direct — the **Future Object** can provide an interface to control the thread, or indirect — if the **Future Object** is deleted before its computation completes, the unneeded computation can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

Example

Consider the metricReport example from the **Result Object (4)** pattern. A **Future Object** could implement the metricReport with exactly the same interface, but with a different performance profile. Using a **Result Object**, the program will wait while the computation is performed. Using a **Future Object**, the computation will be performed in a parallel thread (started by the computeMetrics message), and the program will wait for this thread when it accesses the metricReport **Future Object** — in the diagram, when the sizeOfInterface message is sent. If the creation of the **Future Object** and its first access are sufficiently separated, the metrics computation will not delay the main thread.

Consequences

A **Future Object** protocol is exactly the same 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**. 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 [7]. A **Future Object** cannot avoid the intrinsic problems imposed by concurrency: the program's performance will become less predictable, and harder to debug [16]. Also, this pattern should be applied only when the parallel computation is independent of the rest of the program.

Known Uses

Future Objects are quite common in parallel and distributed computing. They were introduced in Multilisp [11] and have been used in Smalltalk [7] and MUSHROOM [13] amongst many other systems [16].

Related Patterns

Future Object has been described briefly as part of the **Active Object** pattern [23]. A **Future Object** can be seen as a **Proxy** [9] for an object which hasn't been computed yet.

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.

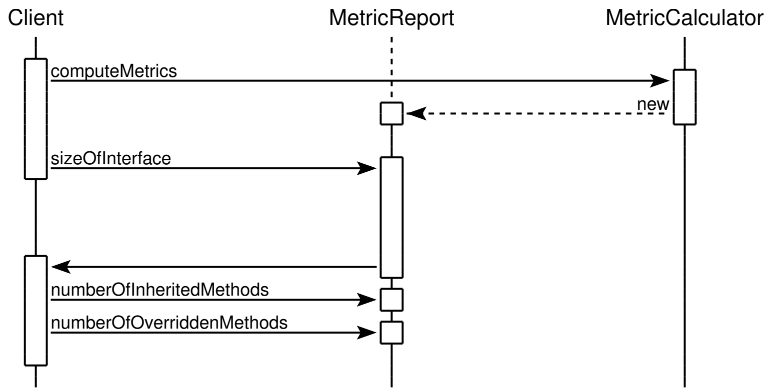
The simplest solution to this problem is for the client to ask the question every time, and then store the results until they are needed. If most of the results are not needed, this will cause a large amount of unneeded computation, and also complicate the client's code, making it harder to write. Alternatively, the server object answering the question could compute and cache all the results, with a 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: *Make a* **Lazy Object** *which 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 should pass the computation's arguments to the **Lazy Object**, which should also cache any information 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 first time the **Lazy Object** is used, it should perform the computation and cache the results. 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. 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.



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 encapsulated 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 need to identify and cache information which the computation depends upon and which might be changed by the rest of the program, and to ensure the computation (when eventually performed) does not itself have side effects on the rest of the program. 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.

The **Result Object (4)**, **Future Object (5)**, and **Lazy Object (6)** patterns are distinguished by 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.

Known Uses

The MUSHROOM system implements a generic **Lazy Object** [13]. The LOOM virtual memory system for Smalltalk used **Lazy Objects** called *leaves* to represent objects which were swapped out into secondary storage [14]. *Design Patterns* describes how ET++ uses **Lazy Objects** called *virtual proxies* to represent large images which may not need to be displayed [9].

Related Patterns

Ken Auer and Kent Beck have described many similar patterns for optimising Smalltalk programs [2].

Acknowledgements

These patterns are revised versions of some patterns from the *Found Objects* pattern language [17], and were inspired by discussions on the *patterns-discussion* mailing list and the *WikiWikiWeb*. In particular, Kent Beck described *Parameters Object* [4] and suggested *Result Object* [3]. Patrick Logan noticed the similarity between these two patterns, David C. Laurence provided the Telecoms example for **Result Object** and Mike Koss described the use of **Result Objects** to handle errors.

Authors and reviewers at the EuroPLOP'96 and PLOP'97 writers workshop provided many pertinent comments and suggestions. Lorraine Body, Linda Rising, and Charles Weir provided detailed comments on the conference draft, as did David Holmes back at MRI and a number of anonymous reviewers. Shepherds Ward Cunningham and Paul Dyson provided many important suggestions regarding the paper's form and content.

References

[1] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The Self Programmer's Reference Manual*. Sun Microsystems and Stanford University, 4.0 edition, 1995.

[2] Ken Auer and Kent Beck. Lazy optimization: Patterns for efficient smalltalk programming. In *Pattern Languages of Program Design*, volume 2. Addison-Wesley, 1996.

[3] Kent Beck. Result object. `http://c2.com/cgi/wiki?ResultObject`.

[4] Kent Beck. Parameters object. Email message sent to the patterns-digest list, March 1995.

[5] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1996.

[6] Grady Booch. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings, second edition, 1994.

[7] Brian Foote and Ralph E. Johnston. Reflective facilities in Smalltalk-80. In *OOPSLA Proceedings*, 1989.

[8] Martin Fowler. *Analysis Patterns*. Addison-Wesley, 1997.

[9] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[10] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

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

[12] Martin C. Henson. *Elements of Functional Programming Languages*. Blackwell Scientific, 1987.

[13] Trevor P. Hopkins and Mario Wolczko. Writing concurrent object-oriented programs using Smalltalk-80. *The Computer Journal*, 32(4), October 1989.

[14] Ted Kaehler and Glenn Krasner. LOOM–large object-oriented memory for Smalltalk-80 systems. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 14. aw, 1983.

[15] Wilf Lalonde. *Discovering Smalltalk*. Benjamin/Cummings, 1994.

[16] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.

[17] James Noble. Found objects, 1996. Reviewed at EuroPLOP.

[18] ParcPlace Systems. *VisualWorks Smalltalk User's Guide*, 2.0 edition, 1994.

[19] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11(3), March 1985.

[20] Alan Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9), September 1982.

[21] Nat Pryce. Type-safe session. In *EuroPLOP'97 Proceedings*, 1997.

[22] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2), April 1986.

[23] Douglas C. Schmidt and Charles D. Cranor. Active object: An object behavioral pattern for concurrent programming. In *Pattern Languages of Program Design*, volume 2. Addison-Wesley, 1997.

[24] Kurt Schmucker. MacApp: an application framework. *Byte*, 11(8), 1986.

[25] Phillip M. Yelland. Creating host compliance in a portable framework: A study in the use of existing design patterns. In *OOPSLA Proceedings*, 1996.

[26] Walter Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design*. Addison-Wesley, 1994.